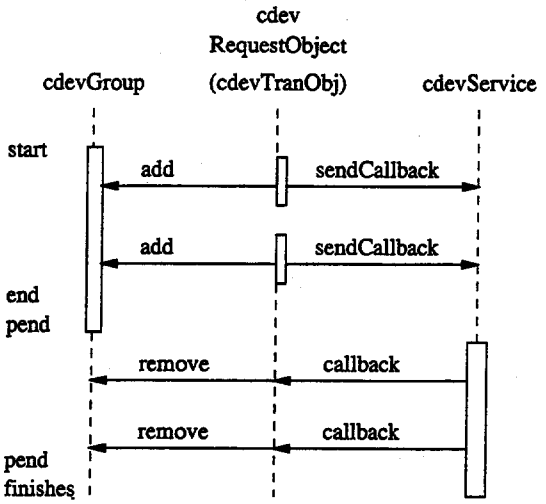


Figure 6: CDEV I/O requests synchronization mechanism

CDEV also allows easy integration of new services. Applications in CDEV only use classes derived from *cdevService* to communicate with services and use I/O request objects derived from *cdevRequestObject* to send out I/O requests in the form of messages. A new service can be integrated into CDEV by developing two new classes. One is derived from *cdevService*, which handles I/O events dispatching. Another is derived from *cdevRequestObject*, which accepts I/O requests in messages and send them out to a service. Once these two classes are implemented and registered into the CDEV name resolution system (see the following section), applications can access the new service without modifying their source code.

4 Related Design and Implementation Issues

Besides the design and implementation issues described in the above section, CDEV also utilizes some common design and implementation techniques to give applications a better interface.

First of all CDEV uses a simple set of enumerated values to denote the status of all CDEV functions. This was chosen rather than using the C++ exception handling mechanism because it will be easier to interface to procedural wrappers for C and Fortran users. CDEV also defines a standard error-reporting class *cdevError* which is inherited by *cdevSystem*. Applications can override the default error reporting function in the *cdevError* class to redirect an error message to an appropriate function.

Secondly, a new data type called *cdevData* is implemented. In order to let applications using CDEV to communicate with different control services or packages, CDEV has to be able to handle different data types in a uniform fashion. Thus a new C++ data type *cdevData* is designed to provide this service. The *cdevData* serves as a repository for data of different types and sizes, accessed by either an integer or character string tag. There is a one-to-one correspondence between integer tags and character tags, so either may be used to insert to retrieve data. Currently a *cdevData* object can hold any type of int, pointer to a character string, char, short, ushort, uint, long, ulong, float, double, or time_val structure and arrays of same. In the future, any structure which is derived from a common parent class can be added or retrieved from *cdevData*. Any applications or new service layers which use *cdevData* for high performance purpose should use integer tags instead of character string tags. Finally, *cdevData* also inlines a lot of its functions to achieve optimal performance.

Thirdly, CDEV contains a name resolution system to locate which service objects derived from *cdevService* to call, and what data to pass to the service in an I/O request. CDEV constructs the name resolution system by parsing an ASCII file that is written by a system programmer in DDL (Device Definition Language) format, and creates a table containing information about devices in a control system. The name resolution system that is called *cdevDirectory* is also derived from *cdevDevice* so that it can be accessed by applications at run-time through the same message based interface. Table 1 summarizes the messages the name resolution system accepts and the results it returns.

Message	Tag Names	Result
service	device, message	service name
serviceData	device, message	data to pass to the service
query	device	list of devices
queryClass	device	parent DDL class
queryAttributes	device, class	all attributes
queryMessages	device, class	all messages
queryVerbs	device, class	all supported verbs
update	value or file	status of the operation
validate	device and class	inheritance relation

Table 1: Supported messages of CDEV name resolution system

Finally, CDEV does not define a new network protocol. It uses the network protocols of its underlying services. For examples, applications using CDEV to access a device via channel access use the channel access protocol.

5 Sample Application

The following code illustrate some of the features of CDEV.

```
#include <cdevSystem.h>
#include <cdevDevice.h>
#include <cdevRequestObject.h>
#include <cdevData.h>
#include <cdevGroup.h>

static void devcallback (int status,
                         void* arg,
                         cdevRequestObject& obj,
                         cdevData& data)
{
    float fval;

    if (data.get (``value'', &fval) == CDEV_SUCCESS)
        printf (``bdl of %s is %f\n'', obj.device().name(), fval);
}



main (int argc, char **argv)
{
    cdevSystem& system = cdevSystem::defaultSystem ();
    cdevDevice& dev0 = cdevDevice::attachRef (``magnet0'');
    cdevDevice& dev1 = cdevDevice::attachRef (``magnet1'');

    cdevData result0, result1;
    int status = dev0.send (``get bdl'', 0, result0);
    // do something with result0
    status     = dev1.send (``get bdl'', 0, result1);
    // do something with result1

    cdevGroup grp;
    grp.start ();
```

```
    status = dev0.sendNoBlock (``get current'', 0, result0);
    status = dev1.sendNoBlock (``get current'', 0, result1);
    grp.end ();
    grp.pend (4.0);
    // do something with result0 and result1

    cdevCallback callback (devcallback, 0);
    cdevGroup grp;
    grp.start ();
    status = dev0.sendCallback (``monitorOn bdl'', 0, callback);
    status = dev1.sendCallback (``monitorOn bdl'', 0, callback);
    grp.end ();
    grp.pend (4.0);

    for (;;)
        system.pend ();
}
```

The main program starts by opening a default *cdevSystem* system that keeps all the information about services and devices. Next, it creates two *cdevDevices*, dev0 and dev1, with name of magnet0 and magnet1 respectively. Then, two synchronous requests are sent out to the devices and followed by a demonstration of the synchronization method of CDEV using *cdevGroup*. Finally, the program enters a event loop in which the attributes bdl of devices are monitored.

6 Benefits of CDEV interface

The CDEV C++ interface provides several software quality factor improvements:

- Correctness: CDEV improves the type-security of higher level applications which no longer need to access low level C-based APIs of services. Strongly-typed object-oriented CDEV interface detects type errors at compile time rather than at run-time.
- Ease to Use and Easy to Learn: the CDEV interface is organized into a set of C++ classes in a hierarchical structure. Hierarchical APIs are typically easier to learn, since their structure indicates closely related operations. In addition, the message-passing interface of CDEV reduces the learning curve dramatically. Simple applications may be written using only 2 classes: *cdevDevice* and *cdevData*. Providing simpler and compact interface allows application programmers to concentrate on design and implementation, rather than wrestling with all the different sets of low level APIs.
- Easy to Maintain: Applications using CDEV are hidden from all underlying services. All I/O requests sent out by applications are mapped transparently on to appropriate services. Furthermore applications are more immune to any changes of the services and are valid even with a newly introduced service without any modification.
- Extensibility: CDEV offers great extensibility to the applications since it is designed using powerful C++ language features (such as inheritance, dynamic binding and polymorphism). Explicitly, CDEV defines several abstract C++ classes which must be inherited by a service that wishes to be integrated under CDEV. With dynamic binding and a name service resolution system at run-time, CDEV can dispatch an I/O request to the new service. Therefore, CDEV is not only a C++ toolkit which only offers predefined C++ classes but also a C++ framework within which new classes can be developed.

7 Concluding Remarks

CDEV is a C++ class library that provides a simple interface to control/DAQ services. It encapsulates an I/O request structure, event demultiplexing and synchronization mechanism of underlying services. It simplifies the development

of control applications by treating all I/O events in the form of messages to devices and the task of integration of new services into CDEV by inheritance , dynamic binding and name resolution. Applications using CDEV will be easy to maintain and to port under any changes of services.

Currently, CDEV has a channel-access service layer, which has all the capabilities EPICS channel access API, tested extensively on HP- UX. There are several applications using CDEV currently in use at CEBAF including one new service, called the model server, which supports DIMAD [9] and other modeling engines. CDEV alone, without the channel access service layer, has been ported to SunOs, Ultrix, VMS and Aix operating systems. The source code for CDEV is available via anonymous ftp from ftp.cebaf.gov in pub/cdev. The web url is http://www.cebaf.gov/cdev.

8 Acknowledgment

CDEV development has been inspired by discussions within the EPICS community. Special thanks to Jim Kowalkowski, Claude Saunders and Marty Kraimer from the Advanced Photon Source at the Argonne National Laboratory for their valuable suggestions and thorough code reviews.

References

[1] Leo R. Dalesio, et. al., "The Experimental Physics and Industrial Control System Architecture: Past, Present, and Future", *International Conference on Accelerator and Large Experimental Physics Control Systems*, Oct. 1993.

[2] William A. Watson III, Jie Chen, Graham Heyes, Edward Jastrzmbski, David R. Quarrie, "CODA: A Scalable, Distributed Data Acquisition System", IEEE Trans.Nuc.Sci., Vol. 41, p61.

[3] Jeff Hill, "Channel Access: A Software Bus for the LAACS", ICALEPCS, 1989.

[4] Bertrand Meyer. "*Object-Oriented Software Construction*", Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1988.

[5] James Rumbaugh. "The life of an object model: How the object model changes during development", *Journal of Object- Oriented Programming*, 7(1):24-32, March/April 1994

[6] Grady Booch, "*Object-Oriented Analysis and Design with Applications*". Benjamin/Cummings, Redwood City, CA, 1994. Second Edition

[7] John Ousterhout, "*Tcl and the Tk Toolkit*", Addison-Wesley, 1994.

[8] Douglas C. Shumidt and Tatsuya Suda. "The Service Configurator Framework: An Extensible architecture for dynamically configuring concurrent, multi-service network daemons", In *Proceeding of Second International Workshop on Configurable Distributed Systems*, Pages 190-201, Pittsburgh, PA, March 1994. IEEE Computer Society.

[9] R. V. Servranckx, "*User's Guide to Program DIMAD*", SLAC Report 285 UC-28 (A), MAY 1985.

Towards a Common Object Model and API for Accelerator Controls

F. Di Maio[a], J. Meyer[b], A. Götz[b,c]
a) PS Division, CERN, 1211 Geneva 23, Switzerland
b) ESRF, European Synchrotron Radiation Facility, BP220, 38043 Grenoble, France
c) HartRAO, Hartebeesthoek Radio Astronomy Observatory, PO Box 443,
1740 Krugersdorp, South Africa

Abstract

An Object-Oriented Application Programming Interface (OO API) can provide applications with an abstract model of the components of an accelerator. The main question is how to encapsulate different control systems into one single abstract model. The abstract model of an OO API can be described in a formal way via object models in order to clarify the semantic issues, to describe the important concepts (device, attributes...), and to decompose the objects up to the granularity where the model of some objects can be shared between labs. A C++ API (as well as C API) can be derived from the object-model. This paper presents a common object model which is derived from both the current CERN-PS model and the current ESRF model. We describe the technical difficulties we encountered in migrating existing control systems into a shared but usable model. We also aim to increase the universality of the model by taking into account the CDEV library, as well as CORBA. A high-level description of the model will be presented with examples of the derived API.

1. INTRODUCTION

The control systems of CERN-PS and the ESRF have, respectively, been rejuvenated and designed during the early 90s. They are both based on objects in the front-end computers. They also have additional similarities due to the fact that they are based on similar technologies and that there has been a regular collaboration between a number of European institutes (SoftCol).

In this framework and with the objective of enhancing software sharing, we asked ourselves the following questions: (a) can we agree on a common API (Application Programming Interface) ? and (b) can we agree on a common model for objects? These questions are particularly pertinent in the context of the current and future OO (object-oriented) technologies, like designing a C++ API or using CORBA technology. CDEV, another candidate for an OO API, can potentially close the gap between object based control systems and the successful EPICS collaboration; it has been included in our analysis here.

In this paper a "draft" model is described, focusing on the technical difficulties identified in this process. It especially focuses on the feasibility of a common model and API. Comments on the sharing process are given by both institutes. Common conclusions are presented at the end.

For more information on the different technologies discussed in this paper, the reader is referred to the following documents: CERN PS equipment access is described in [1] and [2], ESRF device access and device server model are described in [3] and [4], CDEV is described in [5] and two different implementations of CORBA are described in [6] and [7].

2. THE BASIC OBJECT MODEL

The basic object model describes the device object and its associated entities: device class, device attributes and device commands. Shared generic software should be based on the features specified in this basic model. By generic software, we mean software which implements services for any piece of equipment, whatever its specific type is.

One object can have many representations. The same device object can, for instance, have three representations: (a) a description in a file or data-base, (b) one "concrete" instance in a front-end and (c) many "images" objects in the consoles. All the representations can use the same object model, although the implementations can differ.

2.1 Device

A device object implements the services provided by the control system to an accelerator equipment, it has a clearly defined functionality, it is unique and persistent, in the sense that it exists only once in the control system and that its state is not maintained by application programs. A power-supply and a beam position monitor are two examples.

A device has a name which is an identifier unique inside the installation. The devices are classified and have an interface composed of attributes and commands.

2.2 Device Class

Every device belongs to a class which describes equipment with the same functionality. Device objects belonging to the same class have the same interface. Every device class has a unique name inside the institute scope or inside a wider scope (e.g. HEP institutes scope), if some device classes are standardized.

Device classes are organized with parent/child relationships (or inheritance tree); defining a class as a child of another one makes it "inherit" its interface (attributes and commands). Multiple inheritance is necessary, especially for adding generic services to a device class, like inheriting the alarm interface from a "deviceAlarm" class in addition to deriving from another device class.

The device object must implement access functions to the class name and to the description of all attributes and commands. It must also implement the "isA" operation which specifies if a device is an instance of or is derived from a given device class. (cf. IBM's CORBA's "Object" implements the "GetClassName", "GetClass", "IsA" and "IsInstanceOf" functions). In the CERN-PS implementations, the device classes ("equipment modules") are concrete entities which can provide services (i.e. objects).

The device class is not necessarily the C++ class of a device object. For example, the C++ class of a power supply device will be "Pow" on a front-end, while the same object can be an instance of a generic "Device" class on the consoles.

2.3 Device Attributes

The device attributes are the "instance variables" of a device object. There are a variety of synonyms in control systems, like "property" or "parameter". The description of a device attribute is composed of: (a) its name, which is a unique identifier inside the device class scope and (b) the type description of its data. The attribute description should also support a "readonly" qualifier which specifies that an attribute cannot be "set" (e.g. instrument acquisitions).

Device attributes can be simple or composite. As an example, the settings of an instrument can be interfaced with a set of simple attributes, like "gain" or "timing" or with a single, composite attribute "settings". The acquisitions, as well, can be interfaced with a set of simple attributes, like "vertical_position" or, at the other end, with a single, composite attribute: "acquisitions" returning all settings as well as all acquisitions.

Simple attributes must be supported as they provide direct access to equipment values and adapt well to simple, built-in types. Simple attributes are very useful for the integration of external software packages like a spread-sheet or a data presentation package. Unlike composite attribute, they do not imply class-specific data description (e.g. mean value first, then...) or class-specific type definition (e.g. struct {...} PickupAcq).

Composite attributes are currently used in both ESRF (often) and CERN-PS (some classes). This is extensively used by class-specific software (applications which know what kind of device they are talking

to). However, as both systems are based on persistent objects in the front-ends, a simple attributes interface can be added when not present. Otherwise, the composite attributes can be re-defined as compositions of simple attributes.

2.4 Attributes Data Types

It is necessary to define, in the basic model, a "basic" set of types for the device attributes and this set must be supported by all generic software. CERN-PS, ESRF and CDEV have basic sets which are very similar, so a small "basic" set of types can be conventionally defined comprising: (a) scalar types: "float", "double", "int", "char" and "long", (b) strings, (c) one dimension arrays of scalars, (d) one dimension arrays of strings and (e) multi-dimension arrays of scalars (CDEV only).

A run-time representation of the data types (e.g. a value meaning "data is an array of short integers, maximum size is 5") is required in the attributes's description and there are a variety of such representations. There are some equivalencies between the different representations concerning the "basic" set of types, this means that conversions are possible with some additional conventions (e.g. CDEV's "offset" is ignored, CORBA's multi-dimension arrays are recursive sequences of numeric types, etc.). The additions of "class-defined" types (like a class-defined structure for a composite attribute) also depends on the representation of the data types.

2.5 Device Commands

The device commands describe the actions that can be applied to a device. The transactions (described below) execute the devices commands and the exchange of data between the device and the application is described by the device command.

In the reviewed control systems and APIs, the device commands belong to three categories: "get"/ "set" commands, "simple" commands and "send" commands. The "get"/ "set" commands are based on attributes, values are exchanged either from the device or to the device and the data are described by the attribute description. The "simple" commands imply no exchange of data, they implement commands like "on" or "reset". The "send" commands (or "putget") have a more flexible description, they are composed of a command identifier and a pair of data description: data sent and data received to/from the device. The CERN-PS control system implements the "get"/ "set" commands as well as "simple" commands. ESRF uses "send" commands, as well as "simple" and "get"/ "set", as sub-cases of "send".

The "get"/ "set" commands and the "simple" commands must be supported. The "simple" commands are described with a command name, unique inside the device class scope. The "get"/ "set" commands are described with an attribute identifier and a direction qualifier. The "send" commands are described with a command name, unique inside the class scope, and a pair of data type descriptions.

3. TRANSACTIONS

3.1 Transaction

The transactions (or requests) execute commands on (distributed) devices. A transaction is an association between (a) device(s), (b) device's command(s), (c) data object(s) and (d) device error(s). Synchronous and asynchronous forms of transactions are possible.

Synchronous transactions, as they exist in all control systems, send one command for execution to a device and wait for the response.

Asynchronous transactions can be split into three forms: (a) non-blocking (parallel or deferred) transactions, with some synchronizing action on the application side (cf. CDEV's "flush", "poll" and "pend"), (b) transactions triggered by "events" (institute scope identifiers), like cyclic acquisitions synchronized with the "end-of-last-ejection" event (a CERN-PS example) and (c) cache/monitor transactions, where system

software updates local copies of remote objects, with some subscription actions, these are also triggered by events. All forms of asynchronous transactions require a callback function (a reference to an application function), executed on completion of a command execution.

On top of these five basic transaction forms, a possibility for sending commands to a group of devices should be available for synchronous and asynchronous operations. The multiplicity of the transactions is defined by the multiplicity of the devices (one or many) and by the multiplicity of the commands (one or many). The multiplicities of the data objects and of the errors are derived from this multiplicity (cf. below). A necessary convention is that a transaction with many devices requires that all the devices "belong to" a same device class and that this class contains the description of the transaction's command(s) ("belongs to" means "instance of" or "derived from", cf. "isA"). In this case, one command can be applied to a set of devices, while all devices have a common description of the command. These "many device, one class" transactions are extensively used at CERN-PS.

The "event" based transactions, as well as the "cache/monitor" transactions, might not be part of the basic model, because the functionality and the definition of standard events needs further discussion. Synchronous and (non-blocking) asynchronous transactions should be implemented, as they are already part of CDEV. The ESRF and CERN-PS control systems today implement only a subset of these transaction forms.

3.2 Device error

In the reviewed control systems (and in many others, as well), transactions produce an integer error code per device and per command. It is a fact that the definition of these error codes is local to each institute, but these codes should be encapsulated into a device error object to provide uniform error services, such as: name, message, severity and "generic" errors. Error severity (e.g. none, warning, error) is a widely used concept although there exist some variations in its definition. A "generic" error can be defined as an error which has a meaning for generic software; CERN-PS examples are: "communication_error" or "not_implemented" (for this instance). It would be (at least) useful that a single device error object manages a set of devices, to be used by the "many device" transactions. If some device classes were standardized, class-specific errors could be added to the description of the classes.

3.3 Data object

The current CERN-PS and ESRF transactions use data parameters composed of a type description and a pointer to an application's buffer (C API). In the context of an OO API and also for the implementation of generic "services", data objects are necessary in order to provide services like memory management and type conversions.

Data objects are built on a type description which cannot be completely hidden (some services need to ask what is inside a data object created by another service). In addition, data objects will be exported/imported outside the device's services (e.g. data presentation, archiving). These two points imply that the types' representation (cf. above) is a very important issue which requires further analysis.

To avoid problems in data representation and data transfers a standard data type representation format should be adopted. Today's candidates are the XDR (xternal data representation) format or the "type code" description used in CORBA implementations. Implementing a dedicated data format might cause problems in adapting to commercial products later on.

4. ADDITIONAL CONCEPTS

The basic model defines the "common agreement" concepts. Many additional concepts must be added either as an option which is not relevant in every context (e.g. beam) or as an optional refinement of a basic concept (e.g. discrete attributes). Some examples are described here and there are also many additional candidates ("host", "accelerator", "archive", "state attribute"...).

4.1 PPM and Beam

The CERN-PS control system implements PPM (Pulse to Pulse Modulation) which implies that any execution of a device command must specify on which "PLS line" it applies. The PS complex is divided into three different "PLS" (synchronization) systems. Every beam that the complex can produce has, in its definition, a "PLS line" selection for each "PLS" system[8].

This means that additional classes must be defined in the CERN-PS implementation (e.g. PlsLine and Beam). The "standard" transactions described in the basic model can then be implemented by means of a global instances of these classes. In addition to that, transactions with an explicit beam reference are required, as well as additional data in the description of devices ("PLS" selector, "ppm" flag) and in the description of device attributes ("ppm" flag). These are mandatory extensions for the CERN-PS.

4.2 Operational state

The "operational state" is a qualifier whose values are: "on", "off", "stand-by", "warning", "error" and "unknown". The operational states are used at both CERN-PS (for attributes and devices) and ESRF (for devices only). Presentation services can use this (e.g. colours) as well as alarm services (e.g. if "off", don't care about tolerance warnings). This is an important concept (it could be in the basic model) and a mandatory extension for ESRF.

4.3 Discrete Attributes

A sub-category of a device attribute is the category composed of attributes for which the set of values is restricted to a known small set. Such attributes are qualified "discrete" in some control systems, while non-discrete attributes are qualified "continuous". By convention, discrete attributes are restricted to simple, integer, data types. A typical CERN-PS example is a "control" attribute whose values implements "on", "off" and "stand-by" commands. The discrete attributes have their description extended with the description (name + numeric value) of all "legal" values. They support "get" and "set" commands with values' name as a parameter (e.g. "set" the "control" attribute to the "on" value). This is a CERN-PS extension for the user-interface services. An operational state is also included in the description of each values for discrete attributes.

5. APPLICATION PROGRAMMING INTERFACE (API)

A complete model must include a complete API. In fact, an OO API and a formal definition of the object model are two very close things.

In this paper we only give a class diagram for a C++ API (Fig. 1). It uses OMT notation [9]; the classes are represented with their variables and functions; the associations are represented with roles and multiplicity (the big dot means "many"). It is only a "high-level" description of some classes (e.g. no creation/ destruction, no access function, no "dispatching" functions, etc.); the type of the function's parameters are not displayed (the actual type may not even be unique). Variable length signatures have also been replaced with ellipses ("(...)"). In addition, "simple" command has been renamed "exec".

The API includes some design options. In the illustration, the device class' services and data are implemented by means of a dedicated C++ class: "DeviceClass". An alternative is to implement these services as class variables and class methods (C++ "static") in the Device class. Another design option, in the illustration, is to implement multi-device transactions as a class service.

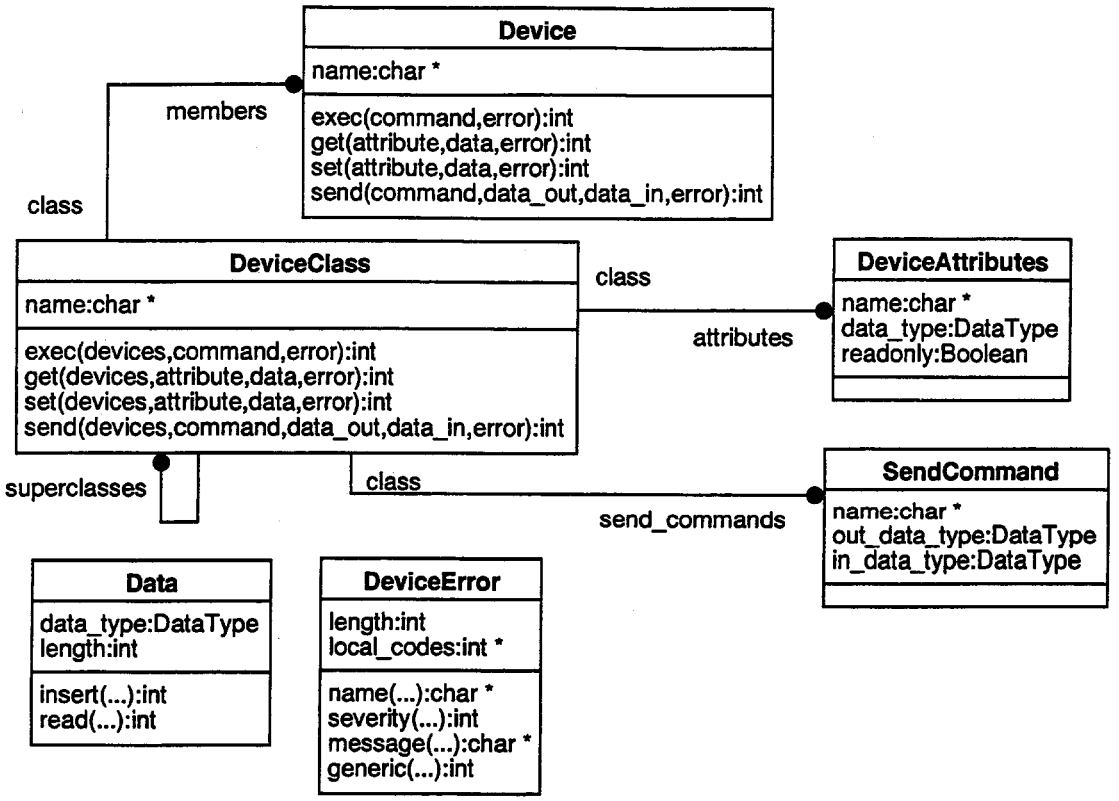


FIGURE 1. Class Diagram

6. CONCLUSIONS

6.1 CERN-PS comments

The model needs to be completed and this should occur in parallel with the identification of "target" applications or services (ones that we want to import or export). On-line modelling may not require, for instance, multi-device transactions, although data-collection services would require these. A particular issue, for us, is to define what "target" services or application use "true" send transactions (i.e. both data out and data in) and for what.

The major constraint we have is that we cannot easily modify the structure of the current front-end objects. These objects are based on attributes and "basic" types and this has a strong influence on our current model. Outside of this scheme, we need to introduce extensions.

We are also extensively using "generic" applications, while "generic", here, means: for any CERN-PS device. Moving these application to a common model would imply some dedicated efforts in the model (e.g. some user-interface extensions) and in the applications. A concrete issue, however, is that the generic applications, as well as the generic services (e.g. data collection), require that the model is clear on the representation of data types, on the data objects and on errors.

As a result of these first concrete activities toward a common model and API, we are convinced that they are highly valuable for the evolution of the console software, that they should continue and that they will, at least, influence any new development at that level.

6.2 ESRF comments

To adapt the ESRF API to the proposed model, is today only possible in a reduced form. With minor modifications, a restricted common API (CERN-PS and ESRF) can be implemented, which is only based on a "get"/ "set" command set, with the basic data types, synchronous transactions and a common set of operational states for devices.

To implement most features of the common model we have to restrict the possibilities of the ESRF control system for some features.

- Today's commands use the "send" format which allows input and output data transfer for the same command. Restricting to a "get"/ "set" command set would require implementing new commands and to use another philosophy.
- Data types to transfer can be defined by the user as part of a device class. To restrict to simple data types would cut this freedom, but it is necessary for sharable devices.

Various extensions are needed to cope with the model.

- Today we do not have asynchronous transactions in the ESRF control system, but they should be implemented in the near future.The outcome of a standard model could guide the development of asynchronous transactions. The implementation might immediately contain all the agreed functionality and standards. We would not need to port it afterwards.
- The ESRF control system also needs to implement a C++ interface on the application side. Today we are using only a C API. A C++ interface would be one of the results of implementing a standard model.
- A new kind of data and error treatment is necessary on the application side, with the use of data and error objects. A mapping to the current system still has to be found.

The discussions on a common object model and API should continue and will influence the software design of new parts of the control system. The application layer is the most promising candidate for sharable software. But the API and the object model must be clearly defined before an application can represent devices of different underlying control systems.

6.3 Conclusions

The implementation of a standard model is possible. But further work has to be carried out. There are many unresolved issues in this version, the major ones being the data objects and the data type representation.

We should study now what kind of software we want to share and adapt the model and the standardisation to the needs in a second iteration.

Evaluating CORBA showed up that it nicely implements distributed objects, but a standard model for accelerator objects must be defined in order to agree on common interfaces.

CDEV is a good example for an API which offers a large range of functionality and is surely a step forward in collaboration. Nonetheless CDEV doubles all object definitions and is restrictive if it is used on top of an object oriented control system. It can be used as a starting point for a common API, but needs further definition of data types, error treatment and additional support for the object model for querying commands of a device or its class hierarchy.

Finally we would like to conclude by saying that the work described in this paper will be continued. The results achieved so far have convinced us that collaborative approaches to object technologies are the way to follow for future accelerator controls.

REFERENCES

[1] J. Cuperus, F. Di Maio*, C.H. Sicard, "The Operator Interface to the Equipment of the CERN PS Accelerators" in ICALEPCS 1993 proceedings, Berlin, Germany, Nucl. Inst. and Meth., A352(1994) pp 346-349.

[2] Franck Di Maio, Alessandro Risso, "The CERN-PS Equipment Access Library" PS/CO/Note 93-87, CERN, Switzerland, 1993

[3] A. Goetz, W.D. Klotz, J. Meyer, "Object Oriented Programming Techniques Applied to Device Access and Control" in ICALEPCS 1991 proceedings, Tsukuba, Japan, Nucl. Inst. and Meth., A352(1994) pp. 514 - 519.

[4] A.Götz, "Device Server Programmer's Manual", ESRF, France, 1993.

[5] Chip Watson, Jie Chen, Danjin Wu, Walt Akers, "cdev User's Guide", CEBAF, USA, July 1995

[6] "AIX Version 4.1 SOMobjects Base Toolkit User's Guide", SC23-2680-01, IBM, October 1994.

[7] "Orbix Programmer's Guide", IONA Technologies Ltd., Release 1.3 July 1995

[8] Julian Lewis*, Vitali Sikolenko, "The New CERN PS Timing System" in ICALEPCS 1993 proceedings, Berlin, Germany, Nucl. Inst. and Meth., A352(1994) pp 91-93

[9] James Rumbaugh et al., "Object-Oriented Modelling and Design", Prentice-Hall, ISBN 0-13-630054-5, 1991

An Object-Oriented Approach to Low-Level Instrumentation Control and Support

J.B.Kowalkowski
Advanced Photon Source
Argonne National Laboratory

ABSTRACT

A common controls requirement is to be able to quickly add support for serial, GPIB, and various data acquisition devices. HiDEOS software provides a means to easily create and maintain low-level device drivers and higher-level control tasks. HiDEOS has an object-oriented task model which hides most operating system details, allowing the user to concentrate on operating the device. HiDEOS imposes a message passing system on the user for interprocess communication, so existing control systems can easily request information and receive results.

INTRODUCTION

HiDEOS is a software package designed as an addition to the Advanced Photon Source (APS) control system. The initial implementation is on the Motorola MVME162 embedded controller, to operate the Industry Pack (IP) Bus and a set of IP modules. Instruments and sensors attached to the IP modules are also operated by the HiDEOS software package. HiDEOS combines parallel processing and object-oriented techniques to produce an operating system shell using the C++ language. The primary function of the operating system shell, in the context of the APS, is to allow a user to easily create, maintain, and integrate device drivers and algorithms into the control system. Many of the targeted instruments have serial or GPIB type communications that require a dialog or protocol.

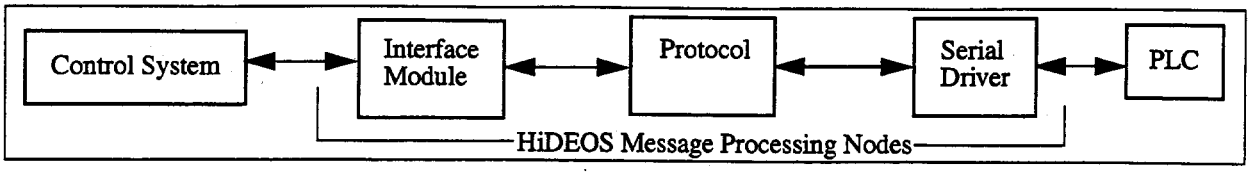
HiDEOS utilizes object-oriented techniques to encapsulate operating system and hardware resources. All components of the operating system are implemented as objects. A user's program is actually viewed as a subclass of the system process. Users interact with the operating system by using methods of the parent class. HiDEOS is a message-driven system, a concept usually associated with parallel processing, where processes get scheduled by the presence of work in the form of a message sent to them. Messages can span CPU boundaries, allowing a problem to be distributed. The package includes a preemptive, multi-tasking kernel for use on a board with no operating system. The kernel is currently capable of running the MVME162 without an additional operating system.

A major goal is to view a running system as a collection of independent message processing nodes, with each node being responsible for a particular piece of equipment, running a protocol, or performing an algorithm. Nodes can find other nodes in the system by using a character string name, attach themselves, and send messages back and forth. A node can be resident on any one of a group of CPUs running HiDEOS. The current implementation requires CPUs to be on the same backplane. An application developer or processing node developer does not need to know which CPU a destination process will be running on. Calls to send and receive messages are the same for processes on a remote CPU or the same CPU. This methodology is illustrated with a real-world problem. A control system must interact with a PLC where the only way to communicate is a serial link. The problem can be logically broken into three sections (see Figure 1), each of which require different system knowledge: an interface procedure to communicate results to and get information from the control system, a protocol driver that can have a dialog with the PLC, and a serial link driver that can actually pump data down and retrieve data from the serial link. With HiDEOS, this problem can be viewed as three separate nodes, linked together with a message pipe. Developers of each module can now solve their own problem, and not worry about the mechanism that will transfer data from one node to the next or which CPU will be running the process.

The current implementation of HiDEOS basically consists of six major components. Together these components allow for basic operating system functionality including interprocess communications. The major components are message management, name service, task management, task dispatching, resource management, and utilities. A typical application running under HiDEOS resides within the task management component. The task is an important and fundamental unit of HiDEOS. The task can make use of board-level services such as tick timers and bus controllers using the resource management component or lock out interrupts using one of the utilities. The task can locate other

tasks in a group of processors running HiDEOS using the name service component. A task can use the message passing facility to locate the destination task and send or receive understood message structures using the message management component.

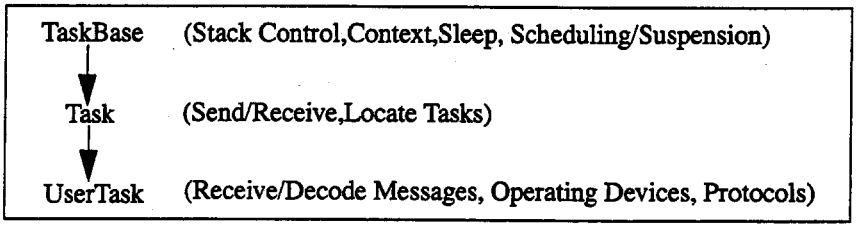
FIGURE 1. Processing Nodes



SYSTEM COMPONENTS

To create a task in HiDEOS, the user must derive a class from the TaskBase class. An actual running task is generated when an instance of the user's derived class is constructed (created). A typical HiDEOS user task has the derivation shown in the class structure diagram of Figure 2. The TaskBase class controls most low-level aspects of a process. A TaskBase instance, to a large degree, is in control of its own destiny. It determines when it should be scheduled to run and when it should be suspended. The TaskBase instance contains the stack and methods to access it. With a setup like this, a dispatcher need only maintain a handle to the TaskBase instance, and manipulate the task through its public interface.

FIGURE 2. Class Structure



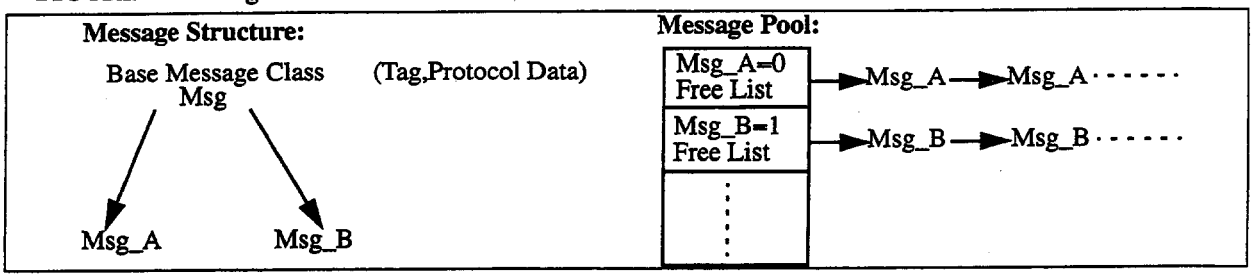
The Task class adds message-passing to TaskBase. The message system will be described later in this document. For this section, the important thing to remember is that the message is a basic unit in HiDEOS that carries information from task to task. The Task class essentially turns the task into an event-driven model. It adds public methods for other task instances to deliver messages to it and the ability for the task instance itself to send and receive messages. In addition, it defines an entry point for user code to be run. When a message is delivered to a task instance, the instance requests that it be scheduled. Some time later it starts running and calls a "Receive Message" function which has been defined by the user. The user is free to run any code in the "Receive Message" function; normally a message will imply a certain action to be carried out. Generally a single task is considered a device driver and is associated with a particular instrument.

Nodes sending and receiving messages from each other require a method to locate each other. The name server component fulfills this requirement. Each CPU in a complete HiDEOS system has one instance of the name server. Each task instance must be given a unique character string name. The name server registers the name with a handle to the task instance. The Task class can be used to automatically register the name of the instance. Any task running in the system can ask the name server for a handle to another task given its name.

Many users of HiDEOS will be interacting with instruments in their tasks. Most instruments hang off of the system bus, so the user must access the instrument by going through the bus controller, which can be thought of as a board-level resource. Board-level resources in HiDEOS are controlled through classes. A resource is any board-level service provided. Examples are the DRAM controller and the bus controller. When HiDEOS starts up, it creates one instance of a specific control class for each of the board-level services available, assuming the service-specific control class has been implemented. The DRAM controller is a good example of a class which operates a service on the motherboard. A user can get a handle to the DRAM instance and ask it for information about the memory, such as "Get Total Available DRAM." The bus controller works in a similar fashion. For the VME bus, the VME bus controller class instance can be asked to enable interrupt levels or open a memory mapping for the backplane.

A typical user application utilizes the message driven capabilities of HiDEOS to retrieve data from instruments on demand. A set of HiDEOS tasks can send and receive a predefined set of messages to each other. The message management system consists of a basic message class from which all user messages are derived. It also contains a message pool which manages user message buffers efficiently. HiDEOS tasks automatically know how to deal with a basic message, therefore any derived message can also be used in the system. Each message type in a complete system is required to be assigned a unique integer tag. The tag is used to generate and free message buffers, and also to identify the true message type in a running program. This is necessary because the interface to the user program is a function of the form "Receive Message" where message is the basic message. It is the job of the user code to check the tag and cast the basic message into its true type. HiDEOS includes utilities to automatically maintain message tags and the message pool. Tags are enumerated during the build process to match the class name with the word "Type" appended at the end to guarantee uniqueness. The user can easily identify messages in the system by checking the type code in the message against the tag enumerations. In Figure 3 below, two tags will be generated: Msg_A=0 and Msg_B=1. When the user program starts running because a message has arrived, the type can be checked against MSG_AType and Msg_BType so the true message can be recognized and the data extracted.

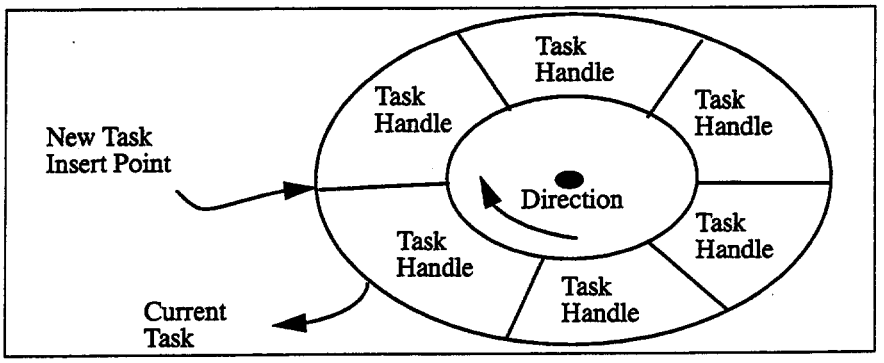
FIGURE 3. Messages



Message buffers must be freed and allocated using the special message pool. There is one instance of the message pool class per CPU running HiDEOS. Message buffers are never released back into the heap. Once allocated from the system heap, they remain in the system and are managed in free lists by the message pool. The message tags are assigned in ascending order, so reusing message buffers from the message pool is just an index into an array of free lists, one list per message type.

HiDEOS contains a dispatcher, shown in Figure 4. As an alternative, the system can easily be set up to use an existing dispatcher which is part of another operating system. An example of where the dispatcher is not used is vxWorks, since it has its own. The dispatcher is extremely simple, it currently does round robin scheduling with only one priority. The dispatcher is implemented as a class. One instance of this class exists for each CPU running HiDEOS. A circular linked list of runnable tasks is maintained by the dispatcher. Each time a time slice is complete, the next task on the linked list is restarted. The TaskBase class is used to add tasks to and remove tasks from the linked list.

FIGURE 4. Dispatcher



MESSAGE PROTOCOL AND DELIVERY SYSTEM

The Task class is an extremely important part of HiDEOS. With all the basic components of HiDEOS introduced, it is now possible to explain the actions carried out by this class for delivering messages. The Task class implements an input queue, and each message delivered to a task in HiDEOS is queued. A task delivers a message to another task by

invoking a "Send" method. A task can wait for messages to appear in the input queue by using the "Wait For Any Message" method.

As explained above, the interface from the Task class to the user's code is through a redefined method in the user's derived class called "Receive Message." The "Receive Message" method is always run in the task's own process space or context, independent of the other tasks running in the system. The user is free to return from this function and should do so when the processing of the current message is complete. Returning from this function automatically implies a "Wait For Any Message" method invocation. At any point in the "Receive Message" function, a call can be made to "Wait For Any Message," "Wait For Message Of This Type," or "Wait For Message From A Specific Task." Invoking any of these can cause the task to suspend itself, removing itself from the dispatching chain until the specified event occurs.

A "Send Message" always executes in the caller's process space or context. The send actually invokes a public interface task of the intended receiver which will place the message in the receiver's input queue. The act of doing so can cause the receiving task to schedule itself depending on its state. If the receiving task is already running, then there is no need to schedule. If it is waiting for any message to appear, then the task will schedule itself as part of the message queuing procedure. If the receive task is waiting for a specific message not of the type being queued, then the task will not be scheduled.

FIGURE 5. Process/Instance Space

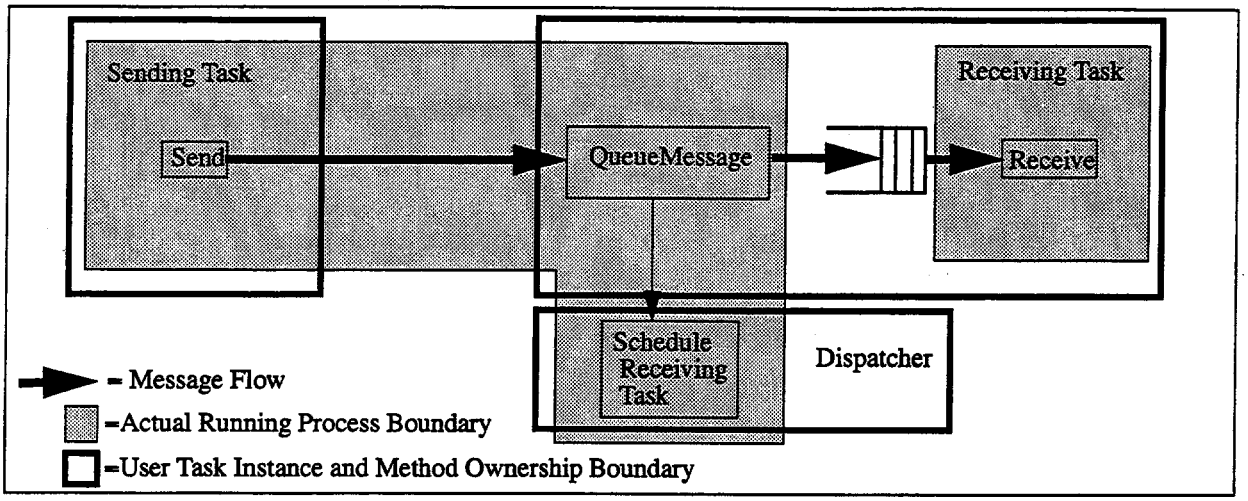
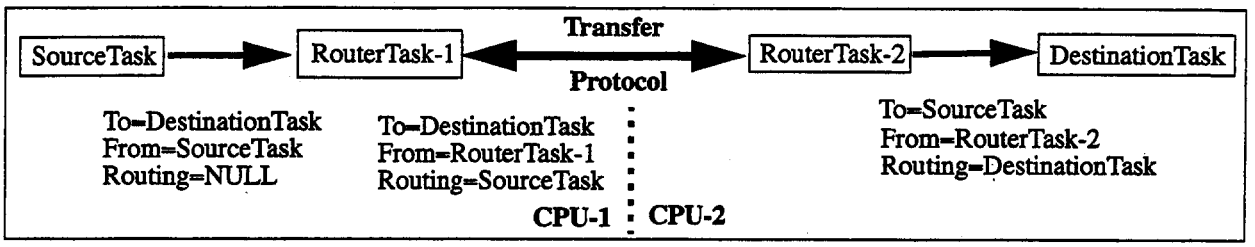


Figure 5 illustrates Task instance bounds and the process or context in which functions get executed. The public interface of a task contains a "QueueMessage" and "ReceiveMessage." The QueueMessage is always executed in the sender's context or process space, the Receive is always executed in the receiver's context.

Messages in HiDEOS use a simple datagram-like protocol. The encapsulation of data within headers is done through subclassing. The Message base class contains the information needed to get messages from one task to another. The class contains the following information: handle to sender's task, handle to receive's task, a special routing task handle for remote communications, and a message type code (tag). The data length is not needed because the type code automatically implies the length of the message. In fact, the message pool can be used to discover the length in bytes of any of the messages given a message type code. As mentioned above, messages are always received in the base class form and must be cast into the correct derived type. A message contains enough information in the header for a user program to respond to the actual sender with results. Allowing two tasks on different CPUs to communicate using the same mechanism is more complex. The third handle in the message, the routing handle, allows for a simple way to transfer messages between two processors (CPUs) transparently. A router typically receives a message from a remote task destined for a local task. The first thing it does is put the from information into the routing handle field, and then place its own handle into the from field. The local task will receive the message from the remote task, do the required processing, and respond to the sender. The real sender is actually the handle in the routing field, but the routing task has fooled the receiving task into sending the message to it. The routing task takes the routing handle and places it back into the to field of the message and forwards it to the real destination.

The message router in HiDEOS is implemented as a HiDEOS task that makes use of the routing field and other flags in the Message base class. Figure 6 illustrates the manipulation of the Message fields by the routing tasks, which catch messages coming in from remote systems and massage the to/from fields to make the message appear to have a source on the local CPU. The routing task also takes on the special responsibility of forwarding name server queries to other CPUs running HiDEOS. Each CPU running HiDEOS has one message router running.

FIGURE 6. Message Routing



To summarize, each instance of the task class in HiDEOS is a separate process or thread. The task class has a public interface with two important functions in it: "Receive Message" and "Queue Message." Other tasks in the system can invoke "Queue Message." Doing so can cause the task in which the "Queue Message" function was invoked to be scheduled to run. The "Receive Message" function is actually user code that is called for each message in the task's input queue. The "Receive Message" function is always invoked within the task instance that owns the input queue in which a message arrived. Messages always appear to be coming from a task on the same CPU, so a user can always reply to the sender of a message.

CONTROL SYSTEM INTERFACE

In order to quickly and easily integrate HiDEOS-based devices into an existing control system, an interface class is available for sending and receiving messages using general user-defined functions. The interface allows for non-HiDEOS programs to interact with HiDEOS programs and to be addressed with a handle just like HiDEOS tasks. It is important to be able to hook HiDEOS into an existing control system, using the existing control system constructs. The interface class allows this to be done. The interface class supplies methods for finding HiDEOS tasks by name and sending messages to them. There is a "Receive Message" call that can be made. This call blocks until a message has been delivered to the interface instance. The interface class provides a way for applications to be event driven. Upon construction of an interface instance, a user event function can be registered that will be invoked each time a messages appears which is destined for this instance. It is up to the user function to do something with the message; usually it will be queued and a second process will be informed that there is work to do. The interface class does not provide any queuing, so it is important that the user event function keep all messages that come in.

A HiDEOS process can receive a message from the interface class instance. The message appears as a message from another HiDEOS task; there is really no distinction between a message from the interface classes and from other tasks. An external process communicating with HiDEOS tasks using the interface class must still retrieve and free message buffers using the HiDEOS message pool component.

One important attribute of using this interface is the ability to create one control system interface for a class of instruments such as ADCs. The interface can specify a message format and protocol which it uses to get information from ADCs. HiDEOS ADC drivers can be created that conform to that protocol. One piece of interface code is capable of talking to many different ADCs, locating them by a character string name.

INITIAL IMPLEMENTATION

HiDEOS has been implemented as an embedded system using the C++ language and the above-outlined concepts. The C++ language was chosen because it is straightforward to understand the generated machine code. It has a simple memory allocation scheme similar to C, which can be used in a very efficient manner by letting HiDEOS manage blocks of commonly used memory. C++ generally produces code in a similar fashion to C, allowing high performance applications to be developed. Complete embedded executables can be produced which do not require any additional run-time libraries. Also, the compiler is available free from GNU.

Most components of HiDEOS were straightforward to implement using simple class hierarchies in C++. However, using C++ with its tight typecasting and lack of true dynamic binding posed several problems with the message passing portion. The C++ class instance creator "new" does not allow the user to dynamically (as the program is running) request a specific type of class instance to be constructed. In other words, the program cannot determine that a "LongMessage" is required and ask the "new" operator to create one. The only argument to the "new" operator is a hard-coded class. This is a problem in HiDEOS because messages require special buffer management so as to not fragment the memory by constantly allocating and freeing messages from the heap. The message pool handler discussed above is responsible for maintaining the message buffers. It is not possible in C++ to create a general "Get Message Buffer Of This Type" function that takes an argument of a message tag. HiDEOS gets around this problem by generating a table of classes and a tag for each class (the type tag). In addition, a case statement is generated (in C++). The case statement has one entry for each integer tag and code which knows how to create a class instance for the given tag (perform the "new" operation).

One outcome of the C++ implementation is a single downloadable HiDEOS image. A completely self-contained HiDEOS system is built for a particular application to run on a given CPU. This is good for embedded applications. What this means is that the downloadable executable will start running as soon as the CPU boots and must have information in it as to what services must be provided or what tasks must be running. The next section discusses the procedure for doing this.

CONSTRUCTING PROCESSES OR TASKS

Developing an application under HiDEOS is a three-step process: decide on a message and message interface that can be used to communicate with the instrument, develop a device driver for the instrument, and create or add instructions to an existing start-up procedure describing how to generate and name the new task.

The main purpose of a HiDEOS message-processing node is to operate a device on demand. Deciding on a message format is very important; it must convey as much pertinent information as possible in one transaction. Several general purpose messages are predefined by the system. Using these messages, if they match, is usually good practice because there are probably a set of other applications that want to communicate with the new task and know how to send and receive the general-purpose messages. Defining new messages usually implies that clients wanting the new services provided by the task must be modified to understand the new messages. An example of a general-purpose message is a "StringMessage." The message passes a generic string of bytes to the receiving task. This message is useful for most serial link instruments. A second is the "LongMessage," which transfers a simple long integer value along with status information. Figure 7 shows an extremely simple example of a user message definition written using C++ syntax. During the HiDEOS build process, the message will be discovered, an integer tag will be assigned, and code will be generated for the message pool to create it, given the tag. An enumerated name "UserMessageType" will also be generated which will be equivalent to the integer tag, and the enumeration will be placed into a global header file of all message tags.

FIGURE 7. UserMessage Definition

```
class UserMessage : public Message
{
public:
  long value;
}
```

The second step in developing an application is to write the user code or driver. An stated earlier, all user programs must be derived from the Task class, contain a constructor to initialize data or a device with which it will be communicating, and have defined a "Receive Message" function to be invoked automatically by the system to process messages. Figure 8 is a simple example of the definition of a HiDEOS task that will use the above-defined UserMessage. The purpose of this example task is to read a register in the address space and return the value back to the

FIGURE 8. UserTask Example

```
class UserTask:public Task
{
public:
   UserTask(char* name);
   void Receive(Message* msg);
private:
   long total_trans;
}
```

requester. The constructor for the UserTask just passes the name to the base class Task, and zeros the total transaction counter:

```
UserTask::UserTask(char* name):Task(name) { total_trans=0; }
```

The "Receive Message" function decodes the message type, casts it to the real derived type, sets the value field from a read of the hardware register, sends the message back to the source, and adds to the total transaction counter. As can be seen, the code fragment only recognizes the message type "UserMsgType"; other messages are passed to the base class Task for processing.

```
UserTask::Receive(Message* msg)
{
   switch(msg->type)
   {
   case UserMsgType: // real type
      UserMessage* m=(UserMsg*)msg;
      m->value=(0xfffffc00); //read register
      Send(msg->from,msg);
      total_transactions++;
      break;
   default: Task::Receive(msg);
   }
}
```

HiDEOS requires a set of instructions to set up processing nodes that are called upon during system initialization. The instructions are contained in a function and are actually a set of user-written C++ statements that can be viewed as a set of start-up rules. These start-up rules specify all the instruments and devices that will be controlled. When HiDEOS starts running, it calls the special user function to create a task instance for each service or device that will be available. Each task instance sits idle after initializing, waiting for incoming messages requesting data transfers from or to the device it owns. The start-up function has the responsibility of creating task instances for devices and giving them names that will be registered in the system. An additional requirement is to hook or link tasks that will be working together, such as a high-level protocol and a serial link task. A typical start-up function contains a series of calls to create and name task instances and a set of statements that connect tasks together is required. The current implementation requires a C++ system initialize hook function. In the future, a parsed rule file will be used to configure a HiDEOS system. This initialization function actually gets called before the dispatcher is started, so processing of messages has not yet begun. Figure 9 shows an example of a function that starts up the task with a name. Other tasks in the system can locate this task by using the name "my-task".

FIGURE 9. User Initialization Function

```
InitializationFunction()
{
   UserTask* ut=new UserTask("my_task");
}
```

CONCLUSION

This paper is designed to be an overview of the underlying concepts of HiDEOS and the methodology used to develop HiDEOS applications. There are many capabilities and details in the real implementation not covered here. For a more complete discussion, including a user's guide, see [1]. Another paper describing the integration of HiD-

EOS into the EPICS control system at the Advanced Photon Source [2] includes a list of supported hardware and several applications using the system, along with extensions required to operate a real HiDEOS system.

ACKNOWLEDGEMENTS

The HiDEOS analysis and design review assistance from John Winans of the Advanced Photon Source at Argonne National Laboratory helped tremendously in producing a solid, coherent product. This work was supported in part by the U.S. Department of Energy, Office of Basic Energy Sciences, under Contract No. W-31-109-ENG-38.

REFERENCES

[1] J.B. Kowalkowski; "Home Of HiDEOS," World Wide Web URL: http://www.aps.anl.gov/asd/controls/hideos/intro.html.

[2] J.B. Kowalkowski; "A Cost-Effective Way to Operate Instrumentation Using the Motorola MVME162 Industry Pack Bus and HiDEOS," these proceedings.

An object-oriented event-driven architecture for the VLT Telescope Control Software

G.Chiozzi
European Southern Observatory, Karl-Schwarshild-Str. 2
D-85748 Garching bei Muenchen, Germany
Email: gchiozzi@eso.org

Abstract

The control software for the Very Large Telescope follows the "Standard Architecture" and is distributed over several workstations, that provide high-level and coordination services, and VME based systems, for real-time control purposes.

The adoption of object-oriented design techniques and the support of a C++ application framework for the implementation of Event-Driven systems reduces considerably the complexity of the applications. The framework provides a general application skeleton and services to automatically receive and analyse events. These are then dispatched to specialized objects, designed to handle specific events.

Since all the run-time and configuration data of the whole VLT is stored in a distributed real-time database, there is a strict coupling between the structure of the database and the applications. As a consequence, the real-time database, although based on the hierarchical model, has been structured to provide support for object-oriented design and implementation.

This paper describes the architecture of the Telescope Control Software (TCS) for the VLT and the object-oriented infrastructure on which it is based.

1 INTRODUCTION

The control software for the Very Large Telescope (described in more details elsewhere in these proceedings[5]) follows the "Standard Architecture"[7] and is distributed over several workstations, that provide high-level and coordination services, and VME based systems, for real-time control purposes. The communication between all the processes running on these machines is based on a message system and a distributed hierarchical database.

This architecture implies that all the applications, but in particular the coordination processes running at workstation level, must be ready at any moment to accept and handle a lot of different kinds of messages, such as new commands, alarms, and notifications from the controlled sub-components.

As a consequence, the design and the implementation of the coordination applications has an high degree of complexity, with an obvious impact on development and debugging time.

The adoption of object-oriented design techniques and the support of an application framework for the implementation of Event-Driven systems reduces considerably this complexity. The framework, based over a set of C++ classes, provides a general application skeleton and services to automatically receive and analyse events. These are then dispatched to specialized objects, designed to handle a specific set of events without having to take into account other parallel but independent conditions.

Given these services, the design and implementation of an application consists of the design of a set of smaller and independent objects specialized in the handling of specific events, such as a command, a change in some database values, the tic of a periodic timer, etc.

Since all the run-time and configuration data of the whole VLT is stored in a distributed real-time database, there is a strict coupling between the structure of the database and the applications. As a consequence, the real-time database, although based on the hierarchical model, has been structured to provide support for object-oriented design and implementation.

This architectural approach has a number of advantages, both in the design/development and maintenance phases; in particular it enforces the respect of standards and provides a mean of sharing and reusing code in the VLT software team.

2 VLT CONTROL SOFTWARE

The control software for the VLT is responsible for the control of the 4 main telescopes and of the auxiliary

ones, the interferometer and the instruments attached to the light beams. The main telescopes can be operated individually or in combination; on an individual telescope multiple instruments can be used simultaneously (one for observation, the others for calibration, preparation or maintenance). This correspond to a distributed environment of 120-150 coordinating workstations and microprocessor based VME systems (Local Control Units, LCUs)[6].

The basic software for the VLT control system, called the Central Control Software (CCS)[3][4], is a large set of modules designed to provide services common to many applications. Most of these common services are available both on workstation and LCU platforms and provide on both the same application programming interface.

The main group of functions provided in the CCS are:

- message handling
- on-line real-time database
- error and alarm handling
- logging
- process I/O
- event monitoring

The real-time database definition language and a framework of C++ classes for the development of event driven applications provide support for object oriented programming. These are described in more detail in the following paragraphs.

3 TELESCOPE CONTROL SOFTWARE

The Telescope Control Software (TCS)[8] controls the main telescopes and the attached equipment that is common to instruments. In particular, for every telescope it controls the main axes (altitude and azimuth), the mirrors, three instrument adapters, the CCDs for auto guiding and active optics, the enclosure and other auxiliary equipment.

The complete VLT will have four equivalent copies of TCS, one per telescope.

3.1 Software architecture

The TCS software can be grouped into four categories:

- user interface
- coordinating software
- subsystem application software
- special interface software

The subsystem application software implements all the functions that can be performed locally in an LCU, without any knowledge of other components of the system. It controls basic actions and motions of subsystem devices such as the main structure of the telescope, mirror supports, adapters and the enclosure.

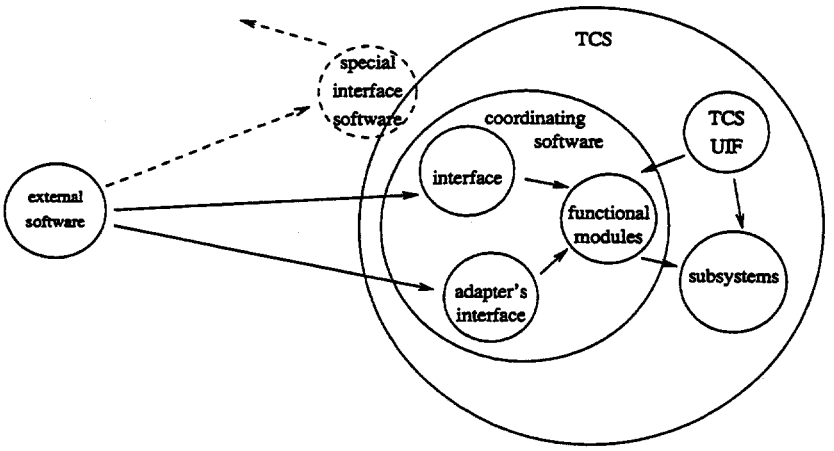


Fig. 1- categories of TCS software

The coordinating software is hierarchically above the subsystem software. It performs actions of combined, coordinating nature, and it often uses one or more subsystems to execute its actions. It runs mainly on the telescope control workstation, although there are major parts running in LCUs.

The special interface software consist of libraries to access, via TCS, external systems such as star catalogues and the Astronomical Site Monitor subsystem.

While the software running on the LCUs has been written in C using traditional techniques, the workstation software is designed using an object oriented methodology and coded in C++.

3.2 Coordinating software

The coordination software provides a public interface for external applications requiring services to TCS and a set of "functional" modules. These are the core modules of TCS software and perform all the coordination work, interacting with each other and with the subsystems.

Some of these modules are for:

- Presetting (setting the telescope to a new target)
- Tracking (maintaining object position following its movement in the sky, without guide star feedback)
- Autoguiding (corrections to telescope position based on guide star feedback)
- Enclosure control (dome rotation, doors, windscreen, louvers)
- Active optics (main mirror axial support and mirror two-position corrections)

All inter-module communication and all communication with subsystems, make use of the CCS message system. Each module has a command interface which is the one that is normally used for all inter-module communication and which is also used by the TCS user interface panels.

3.3 Coordination module's architecture

All coordination modules have the same architecture and their implementation is founded on an object oriented framework based on C++ classes and real-time database classes. Thanks to this approach, maintenance of already existing modules and development of new modules is much easier and, at the same time, VLT conventions and rules are automatically enforced because they are implemented in the base classes used in the implementation.

All the modules are implemented through one or more independent processes.

If a module is made up of more than one process, only one of them is responsible for providing the public interface to the external world, i.e. to receive commands from external modules and to send back the corresponding replies.

The other processes are just slave processes used to perform specific sub-tasks and to improve parallelism.

All coordination modules receive commands from external applications, analyse and elaborate the incoming data and initiate all the necessary actions sending requests to other coordinating modules and to subsystems. They must always be ready to process a new command within a typical "command response time" of 100 milliseconds and many actions must be initiated and handled in parallel as far as possible.

4 THE EVENT HANDLER

In order to meet this requirement, every process is designed in an event-driven way and the implementation is based around the event-handler provided by the evhHANDLER class, which is a core component of the C++ library provided by CCS[1].

Every process contains an instance of this class.

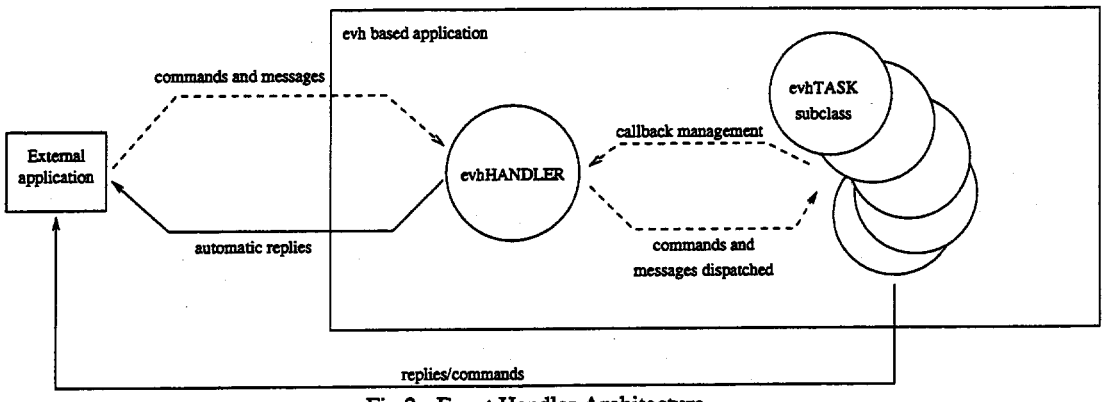


Fig.2 - Event Handler Architecture

This object receives and parses all the incoming events, such as:

- CCS commands
- CCS replies and error replies to sent commands
- CCS time out notifications

- CCS periodic timer notifications
- CCS Data Base Event messages
- UNIX signals
- UNIX file inputs

It then searches an internal database for a list of functions or object methods to be called in order to handle that specific event. Several callbacks, in principle unlimited, can be registered for a single event. For synchronization purposes, it is also possible to define callbacks to be invoked after a combination of events. The callback list is dynamic and the evhHANDLER class provides methods to add and delete elements from the list.

Every list of callbacks must return to the event handler within the "command response time" of 100 milliseconds and the event handler itself takes care of checking this condition, issuing error messages if it is not met (coordinating modules have only soft and not hard real-time requirements).

The design of every process consists mainly in the design of the independent objects providing the methods to be attached as callbacks to the event-handler.

Usually there is one object for every task (a TASK object) that can be executed in parallel inside a process and all the objects have their own independent "life". Whenever an object is waiting for commands or for messages and other events from subsystems, the event-handler is ready to receive events and dispatch them to other objects.

5 THE TASK CLASSES

In order to help to implement the TASK objects and enforce at the same time that the VLT standard rules and protocols are followed, a wide set of base classes are provided.

For example, it is required that every VLT module must be able to handle a certain number of "standard commands". As a consequence an evhSTD_COMMANDS class exist that implements all the standard commands in consistent and reasonable way. A fully compliant VLT application can be built with a few lines of C++ code just allocating an instance of evhHANDLER and one of evhSTD_COMMANDS classes. If a command must be implemented in a different way, the related callbacks can be overloaded or a new object can be allocated for its handling.

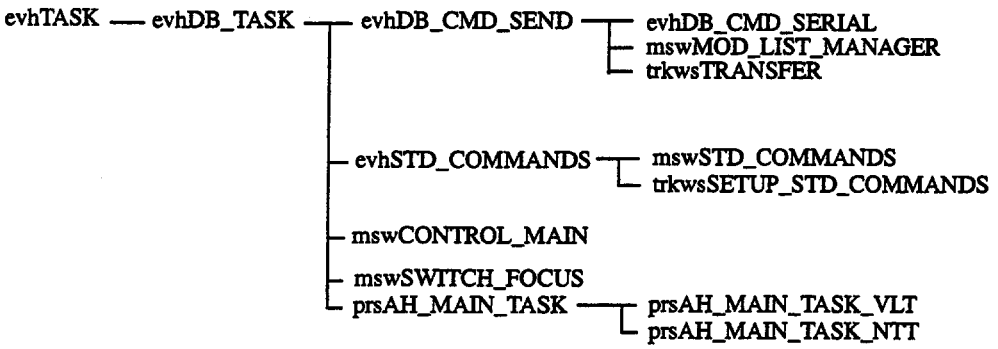


Fig.3 - Some of the classes in the hierarchy of evhTASK

Other classes provide a standard architecture to send commands to one or more external modules and collect the replies. The developer must just derive a sub-class where one or more callbacks are overloaded to specify the object's behaviour when successful replies, error conditions or time-out events are received. This scheme can also handle complex synchronization and queuing protocols, saving a lot of development time and, most importantly, warranting that the same concept is implemented consistently and in the same way everywhere. This is of great benefit for maintenance and debugging.

6 THE REAL-TIME DATABASE

All data that can be of interest for external modules, to get a picture of the status of the system, or for system tuning and configuration, are stored in the real-time database.

This is a hierarchical database distributed on the different workstations and LCUs, mapping the physical and logical objects that constitute the VLT control system and describing how they are logically contained, one inside the other (how one component is "part-of" another one).

Each local database has a partial image only of the units of the system that the particular workstation or LCU has to use. The whole database is the merge of all the local sections.

There will be a lot of different places in the database describing objects with the same or a very similar internal structure and behaviour. For example the VLT system will have a lot of motors, encoders or moving axes that have the same general characteristics.

This description of the VLT real-time database fits very well in an object-oriented model, but the implementation of the database itself is based on a commercial hierarchical database (RTAP, from Hewlett-Packard) that does not support object oriented concepts.

For this reason we have developed a preprocessor and a class browser that extend the semantic and syntax of the RTAP database definition language, introducing the concept of class, with inheritance and overloading[2].

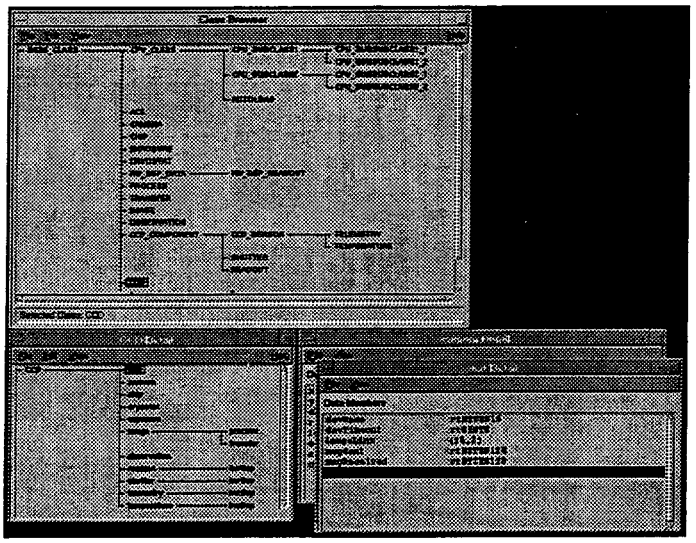


Fig.4 - User Interface for the DBL class Browser

RTAP syntax allows only to define "points" as data structures containing a set of "attributes" of predefined basic data types. The hierarchical structure is built defining a new point as a sub-point of an already existing one. If the same data structure (for example describing a motor) is used in more places, the whole definition of the database branch must be copied.

The extended syntax introduces the concept of "class" as a definition of a new structured data type that can be used as any other available data type. This means that a class instance can be used just in the same way as a native type while defining attributes inside new classes or points.

Each new class has to be derived from another one, from which it inherits all the attributes. Inside a class definition (or a point instantiation) it is also possible to:

- redefine attributes to assign new initial values
- add new attributes
- overload structured attributes

Methods to be used from C++ applications can be implemented developing a C++ twin class, using specific classes provided in CCS.

7 THE TCS DATABASE

Every TCS module defines its own database branch on the workstation's database, following a standard structure. This is automatically imposed by using the database classes coupled with the C++ classes provided by the event handling application framework: for every C++ class accessing the database, there is a twin database class containing the required points and attributes.

The use of inheritance lets the programmer specialize the database class according to the changes in the C++ class.

The application "owner" of the database branch is the only one with write access. All the other applications can read database values by accessing them directly or, better, through the provided C++ access classes.

The following figure shows a simplified view of a TCS database:

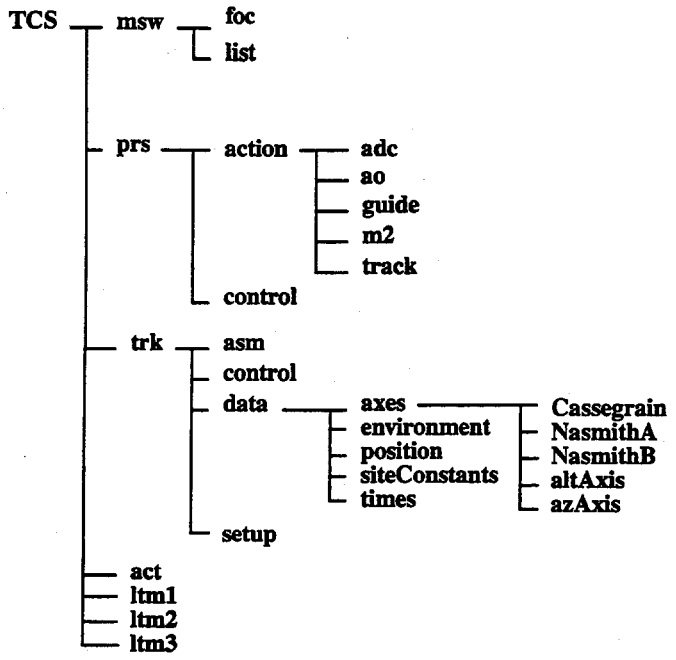


Fig.5 - Structure of a TCS real-time database

8 STATUS

The real-time database preprocessor and the event handling tool-kit have been developed during the second half of '94 and the first half of '95 and have been extensively used in many VLT applications.

In particular they have been used for the development of TCS part 1 (which includes Presetting, Tracking and Mode Switching modules), released in September '95.

TCS part 1 will be tested at the end of this year on the NTT (New Technology Telescope) in LaSilla as part of the upgrade of its control software. The NTT will use as much as possible the same software as the VLT, in order to reduce maintenance resources and to test it before the VLT itself is available. In the case of the TCS, the software has been explicitly designed for this purpose, isolating in specific subclasses all the functionality that is different in the two telescopes. In every case where specific behaviour must be implemented, a base class provides whatever can be in common and two specific sub-classes (one for NTT and one for VLT) implement the part that is unique to each system.

In the first half of '96, there will be tests on a pre-assembled telescope at the manufacturer's site in Milan, Italy. It will be a test of the telescope structure, drive system and encoders and also of TCS part 1.

As a by-product of the development of TSC part 1, a number of general classes have been developed and added to the set provided by CCS. These classes are now used by other applications.

9 CONCLUSION

The adoption of an object oriented design methodology, with the essential support and guide to implementation provided by the C++ application framework, has allowed the development of a Telescope Control System that is intrinsically more maintainable and extendible compared with traditional techniques.

The initial effort spent in designing the general event-driven architecture and implementing the support tool-kit has been already paid back by the fact that the software can be easily adapted to other systems (like the NTT and, in perspective, the auxiliary telescopes). Moreover, the same general concepts are now being applied to other components of the VLT control system, like instrument software.

This approach provides a number of benefits:

- Different applications, developed by different teams, share the same architecture, making easier maintenance in the future.
- Standards and conventions are not only stated "on paper" but automatically enforced by the use of standard classes.

- Important code components are better tested because they are extensively used in many applications.

10 ACKNOWLEDGMENTS

The author wishes to thank all colleagues in the VLT Software Group , and any other ESO staff, who have contributed to the concepts, ideas and software reported in this paper.

11 REFERENCES

[1] G.Chiozzi - CCS Event Handling Tool-kit User Manual - VLT-MAN-ESO-17210-0771, European Southern Observatory, 1995

[2] G.Chiozzi - CCS On Line Database Loader User Manual - VLT-MAN-ESO-17210-0707, European Southern Observatory, 1995

[3] B.Gilli - Workstation environment for the VLT - Proceedings of SPIE, vol.2199, pp.1026-1033, 1994

[4] B.Gustafsson - VLT Local Control Unit Real Time Environment - Proceedings of SPIE, vol.2199, pp.1014-1025, 1994

[5] G.Raffi - Status of ESO/VLT Control Software - These proceedings

[6] G.Raffi - Control Software for the ESO VLT - Proc. Int. Conf. on Accelerator and Large Experimental Physics Control Systems, Tsukuba, Japan, 1991, KEK Proc. 92-15

[7] C.O.Pak, S.Kurokawa, T.Katoh - Eds. Proc. Int. Conf. on Accelerator and Large Experimental Physics Control Systems, Tsukuba, Japan, 1991, KEK Proc. 92-15

[8] K.Wirenstrand - VLT Telescope Control Software, an overview - Proceedings of SPIE, vol. 2479, pp.129-139, 1995

NOTE: The postscript files of the ESO documents, included those mentioned in this paper, are available through anonymous ftp at ftp.eso.org.

Gathering Data from the Fermilab Linac Using Object-Oriented Methodology

Elliott McCrory
Fermilab
Batavia, IL 60510 USA

Abstract. For a number of years, a simple set of objects in C++ has been available to the Fermilab Linac Group for accessing data from the Linac control system. This suite of classes is a simple and powerful way to access this system. The objects are based on the way in which the accelerator data are stored in the local control stations and on the protocol through which these data are transmitted on the network. This paper describes the objects and some of the ways in which they have been used. In particular, several multi-purpose UNIX-style data acquisition tools have been written, along with an interface to pre-existing software packages.

1. Introduction

In order to fully describe this simple set of objects, it is necessary to understand a bit of the environment in which this system operates. First, we will briefly describe the Fermilab 400 MeV Linac, followed by a short description of the control system which this accelerator uses. Then, we get into the description of the objects used on the UNIX consoles for this system: abstraction and encapsulation of the data structures and methods. A few of the applications are described, followed by some operational considerations of this system.

The genesis of this work came under Michael Allen[1]. He performed the basic abstraction and encapsulation of the objects described here.

2. The Fermilab Linac

The Fermilab Linac accelerates negative hydrogen ions (H^-) from the ion source to 400 MeV through multiple stages of acceleration: ion source, 750 keV column, 116 MeV 201 MHz drift-tube linac and 400 MeV 805 MHz side-coupled-cavity linac[2]. There are a pair of (redundant) ion sources, five 201 MHz rf systems, eleven 805 MHz systems, a sub-system for the quadrupole magnets in the 805 MHz portion of the linac and a sub-system for the beam diagnostics. Vacuum is controlled, logically, through the rf sub-systems. This linac cycles at 15 Hz and the ion beam can be accelerated on every rf cycle. The beam pulse length varies from 10 to 60 microseconds. Our average current today is 45 mA, for a total delivered charge of as high as 1.7×10^{13} particles per pulse. With minor modification, this charge could be increased to 3×10^{13} ppp[3].

3. The Linac Control System

The control system for this linac is described in detail elsewhere[4]. The primary design goal of this system (in 1981) was to ensure that if a device went out of tolerance, then beam could be disallowed before the next 15 Hz cycle occurred. In the days before the present control system, beam pipes and vacuum valves had been destroyed by the linac beam[5]. Directly from this primary specification falls the need to have the control system rigidly synchronous to the 15 Hz cycle time. The alarm scan for each linac control station happens at 15 Hz, and about 5% of the analog readings in the linac have been enabled to inhibit beam when they go out of tolerance.

The software architecture[6] on each linac control station is loosely coupled to PSOS. Communication from a console computer is established through UDP/IP sockets, using either the Fermilab Accelerator Controls NETwork (ACNET) protocol[7] or through a custom "Classic Protocol."

The linac control system is distributed to seventeen VME-based crates containing MC68020 processors (which could be upgraded to 68040's), communicating with each other and with the outside world via Token Ring. Each of the seventeen stations controls up to six Smart Rack Monitors (SRM) [8], which contain the D/A's, A/D's and digital I/O necessary to talk to real equipment. The VME crate can also contain digitizers, but the only digitizers presently used on our VME stations are 1-to-10 MHz "quick digitizers" [9] for looking at the transient beam diagnostic signals. Each local control station owns approximately 400 scalar, analog devices.

We have recently developed an "Internet Rack Monitor" which is a stand-alone rack-mounted chassis which contains a MC68040/Ethernet/Industry Pack VME card, in addition to the D/A, A/D and digital I/O of an SRM [10].

This control system is used at other locations: the Fermilab D0 experiment, TESLA, Michigan State nuclear accelerator, Fermilab Booster and Main Ring High-Level rf, the Loma Linda Cancer Treatment Facility and the Shreveport PET-isotope production facility.

4. UNIX Data Acquisition and Control

The data acquisition and control software for the Linac consoles runs on Sun SPARCstation computers running Solaris 2.4. It has, in the past, run on SunOS 4.1, SunOS 3 and on the 68020-based MassComp computers of 1988. The system is implemented in C++. The ideas presented here are evolving as the C++ definition evolves and as our experience with objects grows. A FORTRAN interface is maintained, minimally.

5. Abstraction of Data and Operations

The objects in this OO system are abstractions of the data types which are present in the local control stations. These data types are:

Scalar, analog data, the associated binary status and control alarms information and database information associated with a real or derived local device;

Binary data, it's alarm and database information;

VME memory, including vectorized analog data associated with the quick digitizers;

Data streams (a generic way to assemble structures of data in the local station);

The means for gathering these data are encapsulated into the objects's methods. These methods include the network protocol for getting the data, database name resolution for analog devices, synchronized data return at a given rate (up to 15 Hz) and synchronized return of data when a specific Tevatron Clock (TCLK) event is received.

6. Description of the Objects

6.1 Major Objects

The major objects used in this system are described here. Refer to Figure 1 as a guide.

`TRAccessObject`: Parent object of all the data acquisition objects. A closely-related class, `CtlMsg`, handles the information necessary to insure that each object gets the proper information from the network.

`BinaryDatum`: Handles the retrieval and processing of the local station's binary data.

`RemoteMemory`: Handles the retrieval and processing of a local station's VME memory.

`DataStream`: Handles the retrieval and processing of the local station's data stream information.

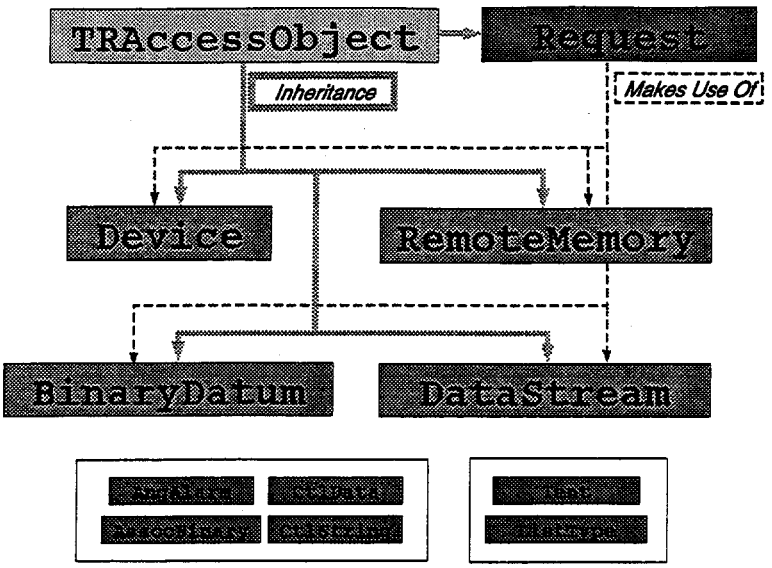


Figure 1, Simple view of the interrelationship of the objects.

`Device`: Handles the retrieval and processing of the local station's scalar analog devices. This class includes: instances of the `CtlData` class for handling the scaling factors on the readings and settings of the device; an instance of the `AngAlarm` class for determining the alarms status of the local device and a copy of the local database information (which can be modified and downloaded to the local station, if desired). Options on this object include: Setting, motor control and associated binary status and control (through `AssocBinary`).

`Request`: Handles the synchronized, repetitive reception of data from the local control stations. This repetitive return is initiated by a single network message. This class includes: Lists of `Device`, `RemoteMemory`, `BinaryDatum` or `DataStream` through the list template `PList<object>`; an instance of `PList<ListType>` to define the type of data returned on the request; an integer representing the period of the return data (1=15 Hz; 30=0.5 Hz, for example); and, optionally, a TCLK event for returning data only on that event. The control of when to return data is handled exclusively by the local station.

6.2 Minor Objects

The minor objects used in this system include:

`ChanIdent` and `AddrIdent`: The logical address of the information within a local control station and on the network.

`AngAlarm`: Handles the analog alarm information for a Device.

`AssocBinary`: Handles the associated binary status and control for a Device.

`CtlData`: Handles the conversion of internal 16-bit data to voltages and to engineering units.

7. Some Applications

We have written a few dozen applications on these objects, and a few of these are summarized in Table 1.

8. Operational Considerations

These classes were derived through the effort of M. Allen in 1988 for the Loma Linda Medical Accelerator, under development at that time at Fermilab. The author has expanded and enhanced these object gradually over the years. R. Florian has contributed a significant number of application programs over these years. In summary, there has been only a minimal effort put into this fairly capable system, no more than 1 person-year.

We recently tested the robustness and throughput of the system on a SPARCstation 2 computer from Sun. This computer was able to flawlessly capture and display over 500 network frames per second using this system.

Application	Description
ac-get-data	General data acquisition on the UNIX Command Line
	Correlation Plots, gating, triggering, and ~ 30 other options
DataViews	Data acquisition interface to the commercial product DataViews
Synoptics	Several synoptic displays implemented using DataViews
checklist	A suite of ~10 shell scripts which runs each day to inform staff of any unusual situation in the Linac control system
	For example: check system date on each local control station, check that some of the important devices acutally are in the alarm scan, check for some rare fault conditions, etc.
Plot package	Using TCL/TK/BLT, scalar and array plot packages
page-g	I/O with the 40x20 dumb-terminal used to access configuration parameters for each local station
IP Node Test	Test that all IRMs are on the network
RDATA Edit	Edit the tables in the local control stations which control the operation of these systems.
Save Restore Data	Edit the analog data description tables in the local control stations.

Table 1, List of some of the applications used in the Fermilab Linac

This system is fully multi-user and multi-tasking.

9. Conclusions

Using an object-oriented approach to data acquisition in the Fermilab Linac has been a direct, simple and powerful way to encapsulate the complexity of data acquisition for this accelerator. New ideas will be implemented as the C++ definition changes and as we become more familiar with them. In particular, templates have not been adequately exploited, multiple inheritance is not used and no polymorphisms have been necessary. This system is evolving.

10. References

[1] Present address: Motorola Corporation, Arlington Heights, IL, email: allen@cig.mot.com

[2] E. McCrory, "The Commissioning and Initial Operation of the Fermilab 400 MeV Linac," Proceedings of the 1994 Linac Conference (Tsukuba, Japan), pp 36-40.

[3] M. Popovic, et al., "Possible Upgrades to the 400 MeV Linac," Internal note.

[4] E. McCrory, R. Goodwin, M. Shea, "Upgrading the Fermilab Linac Control System," Proceedings of the 1990 Linac Conference (Albuquerque), pp 474-476; `http://adwww.fnal.gov/LINAC/linac.html`

[5] Private communications with Charles Schmidt, Linac Group leader.

[6] `http://adwww.fnal.gov/LINAC/software/locsys/locsys.html`

[7] C. Briegel, et al., "The Fermilab ACNET Upgrade," NIM A293 (1990), 235-238.

[8] `http://adwww.fnal.gov/LINAC/hardware/srm/srm.html`

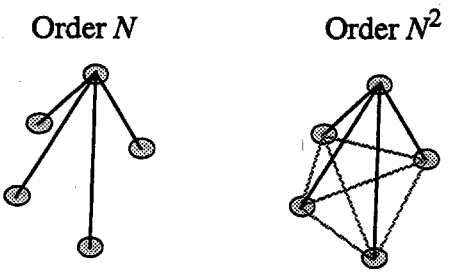
[9] `http://adwww.fnal.gov/LINAC/hardware/quickdig/QuickDigitizerDoc.html`

[10] `http://adwww.fnal.gov/LINAC/hardware/irm/irm.html`

Distributed Computing Environment in the ESnet Community

R. Roy Whitney
Continuous Electron Beam Accelerator Facility
12000 Jefferson Avenue
Newport News, VA 23601
whitney@cebaf.gov, http://www.cebaf.gov

The Energy Sciences Network (ESnet) Community is undertaking an ambitious project to coordinate its distributed computing. A reasonable question is: *"Why Coordinate Distributed Computing?"* The answer can be seen in the figure below:



The issue is complexity.

Without a coordinating element, the number of links for communicating between a number of nodes would grow with order N^2. Inserting an effective coordinating principle can reduce the number of independently managed links to order N while still leaving all of the links available for actual communications. A good example from telecommunications is the public phone system where we rely on central coordination of the phone numbers rather than having to personally negotiate the switching information with everyone with whom we wish to communicate. The Internet's distributed system of name servers is another good example.

The meta-question to the first question is: *"Why use distributed computing?"* The answer can be put in three parts:

- Individual end users require access to far more information and computational resources than they can acquire locally.
- For some applications, distributed computers are nearly as efficient as centralized computers and the total CPU resources available may be significant.
- Many resources are unique. They cannot easily be replicated. Some examples in the DOE community include research accelerators, fusion facilities, systems of virtual caves, and environmental laboratories.

The overall response to the questions can be summarized in the concept of virtual laboratories. Humans want their resources responding within a second and at arm's length or closer. Using coordinated distributed computing, these requirements can be met by placing facilities on-line via Internet style technologies. Further, human communication is 10% verbal, 40% audio and 50% physiological. Using combinations of words, sounds and pictures including animation, virtual laboratory technologies can exploit all of these aspects of human communication. The human end user receives the benefit of virtual presence at on-line facilities and makes use of a rich collection of resources.

The ESnet Community has three main sources for its distributed computing vision, requirements and technological capabilities:

- End users
- U.S. Department of Energy (DOE) /Energy Research (ER) /Office of Computing and Technology Resources (OCTR) /Mathematics, Information and Computing Sciences Division (MICS)
- ESnet Steering Committee (ESSC)
  - ESnet Site Coordinating Committee (ESCC)
  - Distributed Computing Coordination Committee (DCCC)

The end users, DOE/OCTR/MICS and ESSC focus on the vision and requirements. The ESCC and DCCC focus on the technological capabilities and implementation.

The current ESnet distributed computing project is Distributed Informatics, Computing & Collaborative Environment (DICCE). The goals for this project are to:

- Set up DICCE for the ESnet Community
- Create the basis for:
  - Virtual Laboratories
  - Networked Collaborative Environments
  - Facilities On-line
  - Distributed Computing Environments

The minimum DICCE is:

- High quality Wide Area Network (WAN)
- Key distribution system
  - Kerberos and public
- Authentication and authorization services
- Open Software Foundation's Distributed Computing Environment and Distributed File System (OSF's DCE & DFS)
- Secure MIME compliant E-mail
- Secure WWW technologies

While most of the funding for DICCE comes directly from the participating facilities, in 1995 DOE/OCTR/MICS provided some funding for a DCCC coordinated set of proposals for DICCE to be used for Research and Development (R&D) activities. More information can be found on these proposals under http://www.es.net. DICCE will be a multi-year project as it builds infrastructure essential for the virtual laboratory complex.

The first requirement for meeting the minimum DICCE is to have a high quality network. ESnet provides T3 (45 megabits/sec) Internet-style connectivity to all of the major DOE/ER laboratories, a variety of other DOE sites, and several DOE centers at universities. Additionally, ESnet provides these facilities with connectivity to the Global Internet via connections to the major commercial Network Service Providers and many regional networks. ESnet was the first major backbone piece of the Global Internet to provide these services using Asychronous Transfer Mode (ATM) technologies. Some portions of ESnet will be upgraded to OC3 (155 megabits/sec) in the near future. ESnet is managed by Jim Leighton at the National Energy Research Supercomputer Center (NERSC). Currently ESnet is moving with NERSC from Lawrence Livermore National Laboratory to Lawrence Berkeley National Laboratory.

The overall DICCE project is managed and coordinated through the ESCC and DCCC via task forces and working groups. The following is a listing of the DICCE-related ESCC and DCCC groups along with information on the ESSC:

- ESnet Steering Committee (ESSC) — Sandy Merola (LBNL) Chair
  Provide programmatic goals, vision and user input to ESnet management.
- ESnet Site Coordinating Committee (ESCC) — Roy Whitney (CEBAF) Chair
  Coordinate site issues for the effective implementation of ESnet at the sites.
- Distributed Computing Coordinating Committee (ESCC) — Roy Whitney (CEBAF) Chair
  Coordinate the implementation of ESnet Community distributed computing. Provide project management for DICCE.
- Key Distribution TF — Bill Johnston (LBNL) Chair
  Set up key distribution for Kerberos V5, OSF/DCE and Public Keys.
- Authentication TF — Doug Engert (ANL) Chair
  Set up cross realm Kerberos V5 authentication ESnet Community wide.
- IPng WG — Bob Fink (LBNL) Chair

Set up IPng open test bed. IPng is critical to long-term DICCE goals when extended to National and Global Information Infrastructure activities.

- Network Monitoring TF — Les Cottrell (SLAC) Chair
  Provide tools for LAN and WAN monitoring.
- E-mail TF — Mark Rosenberg (LBNL) Chair
  Institute MIME, PEM and PGP compliant E-mail ESnet wide. This technology will also serve as the basis for digital signatures.
- Remote Conferencing WG — Kipp Kippenhan (FNAL) Chair
  Advance collaborative video conferencing.
- Distributed Computing Environment WG — Barry Howard (NERSC) Chair
  Coordinate implementation of OSF's DCE, CORBA, etc.
- Andrew/Distributed File System TF — Troy Thompson (PNNL) Chair
  Set up an ESnet Community-wide file system.
- Distributed Systems Management WG — John Volmer (ANL) Chair
  Provide structure for implementing systems management in the ESnet DICCE.
- Applications WG — Dick Kouzes (WVU) Chair
  Coordinate DICCE applications development including underlying tools.
- Group Communications WG — Allen Sturtevant (NERSC) Chair
  Coordinate file types supported in the ESnet Community for ftp servers, WWW, MIME, etc.
- Architecture TF — Arthurine Breckenridge (SNL) Chair
  Consider high-level issues for setting up DICCE throughout the ESnet Community.

Clearly a significant level of resources isbeing focused on the distributed computing infrastructure at the DOE facilities. It is appropriate to comment on one project of particular interest to the ICALEPCS community. The DCCC DICCE proposal was actually made up of 23 separate proposals. DOE/OCTR/MICS approved nine of these proposals and funded seven. The 23 proposals were self-rated by the DCCC proponents. The number two rated proposal was that from the Experimental Physics and Industrial Controls System (EPICS) collaboration to put a DICCE layer around EPICS so that facilities running EPICS could be brought on-line via the Internet in a secure fashion.

As previously noted, one of the DICCE goals is to provide a global file system available to the entire ESnet Community. Presently, Argonne National Laboratory (ANL), NERSC/LLNL, Pacific Northwest National Laboratory (PNNL), and Sandia National Laboratory (SNL) have OSF/DCE including DFS on-line in a cross-realm configuration, i.e. a simple change directory command moves the user from the file system at one facility to another. Ames Laboratory, CEBAF, LBNL and several universities will also soon join the file system. The goal is to move this DICCE pilot to a production service for the ESnet Community distributed collaborations by Summer 1996. Note that meeting this goal implies that key distribution systems have been made operational, a significant challenge.

Higher level services are anticipated by Fall 1996. Long-term DICCE milestones include providing a "Single Environment" for ESnet Community collaboratory end users, a diverse set of virtual laboratory tools and applications, and all ESnet Community facilities on-line where applicable.

Additionally, since the start of the DCCC DICCE project the World Wide Web has switched from being driven by R&D interests to being driven by commercial interests. For example, the technologies for authentication and authorization on the Web are being determined by banks and credit card companies. As the Web is proving a versatile virtual laboratory tool, the DICCE project is busily incorporating these Web offerings. The end user will require only one Web browser to securely interface with and control on-line facilities, data, and other budget, personnel and management information.

Consider this prediction: Your successful R&D laboratory will aggressively deploy mission-oriented virtual laboratory/DICCE technologies; complexity will be coordinated and your overall organization will have a higher productivity. As you visit different facilities or communicate with your colleagues, carefully check what is happening with network access to the facilities, scientific and engineering data and information, and management information system resources. Check to see and hear how the more successful projects are achieving their goals.

In summary, because ESnet is a mission-oriented network, its community can directly attack the R&D challenge of setting up the distributed computing infrastructure for virtual laboratories in a coordinated manner. Its scientists and engineers work together to meet their programmatic requirements. Many university researchers are also involved. It is anticipated that many of these ESnet distributed computing projects will have significant impact far beyond just the DOE environment — reminiscent of how the European Center for Nuclear Research's (CERN's) World Wide Web has impacted society. Distributed computing and virtual laboratories will be essential elements of the National and Global Information Infrastructure.

A SERVER-LEVEL API FOR EPICS

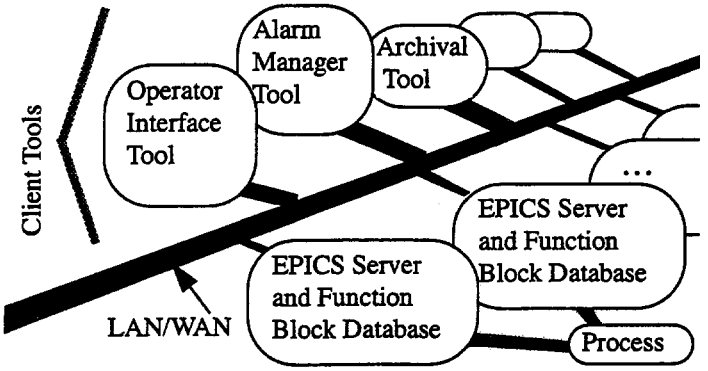
J. O. Hill
LOS ALAMOS NATIONAL LABORATORY GROUP AOT-8
MS H820, LOS ALAMOS, NM 87545, USA

The existing Experimental Physics and Industrial Control System (EPICS) applications programmers' interface - Channel Access - has in our experience been a catalyst for efficient collaborative software development. Having seen real cost and quality benefits resulting from the adoption of modular system design techniques in a control system context, we propose a new applications programmers interface (API) for EPICS to be installed just beneath the existing channel access server. This new API will encapsulate the EPICS IO system making it another modular, replaceable software component. We will thereby eliminate several existing EPICS limitations including: only one choice for the front end operating system; only one EPICS front end architecture; difficulties exporting process variables from client-side applications and difficulties creating transient process variables. Potential applications are gateways between existing control systems and the expanding EPICS tool set, gateways between non-essential users and the live control system, access to alternative data stores such as commercial databases and light-weight IO controller implementations. We believe that this API will result in greater freedom to pick and choose components of EPICS and ultimately a wider application of EPICS.

1.0 INTRODUCTION

The Experimental Physics and Industrial Control System (EPICS) is a process control and data acquisition software toolkit in use at a number of sites world wide. The software was designed for general utility and has been successfully installed into a wide range of applications including particle accelerators, experimental physics detectors, astronomical observatories, municipal infrastructures, petroleum refineries, and manufacturing. A scalable, fault-tolerant system that follows the "standard model"[1] can be created with the toolkit. Compilers and filters are used to instantiate control algorithms in front-end computers from function block and state-machine formalism-based input. EPICS communication occurs within a software layer called channel access (CA) that follows the client-server model and employs the internet protocols (Figure 1). A mature set of client-side tools provide operator interface, alarm handling, archival tasks, backup, restore, state sequencing, and other capabilities. Client-side interfaces are provided to commercial packages such as IDL, TCL/TK, MATLAB, and Mathematica. There is also an expanding library of hardware device drivers that have been written for use with EPICS. Recently we have seen a number of sites working on generic physics and control theory applications that will interface directly with EPICS. All of these components taken together form a toolkit that allows control system installation while writing a minimum of low level code.The details can be obtained on the world-wide web[2] and from previous papers[3][4][5][6]. EPICS is very unusual among control system software packages in that it has been developed by a collaborative effort of several laboratory and industrial partners[7].

FIGURE 1. EPICS



2.0 PRESENT LIMITATIONS

There are at present several limitations that we would like to remove from EPICS. Currently, the EPICS client-side tool set only interfaces with the EPICS IO function block database. We believe that the generic nature of the EPICS client-side tools and the channel access protocol should permit their use in a wide range of applications. For example, there is no compelling reason why the client-side tools could not be used unmodified with the many present and future control systems developed at other labs. When designing the software for a control system there are trade-offs required between maintaining stability for installed systems and allowing evolution for new projects. For instance, we can foresee many potential improvements to the EPICS IO function block database but are reluctant to make abrupt changes because we must maintain stability for the many existing EPICS installations. Modest architectural changes to EPICS would make it possible for present and evolved versions of the IO function block database to coexist in the same control system, thereby allowing rapid evolution of new capabilities while maintaining stability.

Another limitation results when components from the client-side tool kit must compute an intermediate result. For instance an operator screen might command another tool to initiate an emittance scan and then display the result when it completes. Where should this result be stored? Presently within EPICS all communication of this nature inevitably occurs through "soft records" in the IO function block database. There are several problems with this solution. One of these is that if two operator screens initiate an emittance calculation at almost the same time but with different input parameters then one operator will be required to wait until the other operator's emittance calculation is completed. Worse still, if mutual exclusion isn't built into the emittance calculation program the operators risk a collision where their answer may not result from the input parameters specified. Another negative is that resources are consumed in a front end IO controller for storing and updating emittance calculation parameters when this may not have been that processor's original purpose. Our conclusion is that these intermediate parameters are not now stored in a natural location within the client server hierarchy. If, when it was appropriate, the parameters were stored within the tools themselves, we would see improved interconnection within the EPICS tool set and less load on critical front end machines.

Note that some of the above "limitations" are not limitations unless you are committed to a tool-based approach. We could interface the client-side tools to other control systems by making large source code changes in them at each site that they are used. We could rapidly evolve the software without regard for the stability required by installed systems. We could also combine existing tools together by making application-specific source code changes each time that they are used in a different combination. However we have not chosen this path because we would not benefit in the long term from the labor of individuals at other sites unless software developed at one site can be used at another without the need for site-specific source code changes.

We believe that a distributed system design employing direct "single-hop" connections between individual hosts has advantages related to improved latency, dispensation of load, and fault tolerance. However there are low priority clients of the control system for which the above factors are less important. For these low-priority clients a more important concern may be to minimize load on critical servers in the front end IO controllers. Each client connecting to a particular server drains resources from that server and the network. Modest architectural changes would allow multiple low-priority clients to share one connection to a server in a critical machine. These changes would guarantee that low priority clients do not overload a critical server or network, and therefore compromise critical functions in the control system. In this way unrestricted use of the system by low priority clients could be safely implemented.

3.0 COST AND QUALITY

When developing software there are many metrics to maximize in order to obtain the best design. However the dominant factor related to the cost and quality of a software product is the size of the software distribution. Increased distribution always lowers the per site costs. Increased distribution also decreases the probability of becoming the site that discovers (and therefore isolates) an esoteric bug. This leads to the conclusion that increased distribution also

improves quality. Our overriding challenge as software designers is to identify common patterns of usage and adopt a common infrastructure so that we maximize the number of users of each and every line of code. Therefore to minimize cost and maximize quality it is necessary for us to evolve the architecture of our systems as we see new opportunities for common infrastructure.

4.0 A NEW SERVER-LEVEL API FOR EPICS

A new server-level application programmer's interface (API) has been added to EPICS. Previously the EPICS server was only able to communicate with the EPICS IO function block data base and this was the only source and destination of data in the system. A diagram depicting the evolution of the software architecture is shown in Figure 2. With a server API it is now possible for EPICS to communicate with alternative data stores in addition to the function block database. Note that on the left side of the figure the server source code and the IO function block source code are tightly coupled. This results in a lower utility of both components. On the right in the figure the server is packaged as a library with a carefully designed API. This allows the server library to be used not just with the function block database but also with many other software components. We will refer to these new components as server-side tools. This small modification to the system architecture opens up a number of new server-side possibilities and therefore proportionately increases the utility of the existing EPICS client-side tool set (Figure 3).

FIGURE 2. SOFTWARE LAYERING WITHIN EPICS BEFORE AND AFTER A SERVER-LEVEL API

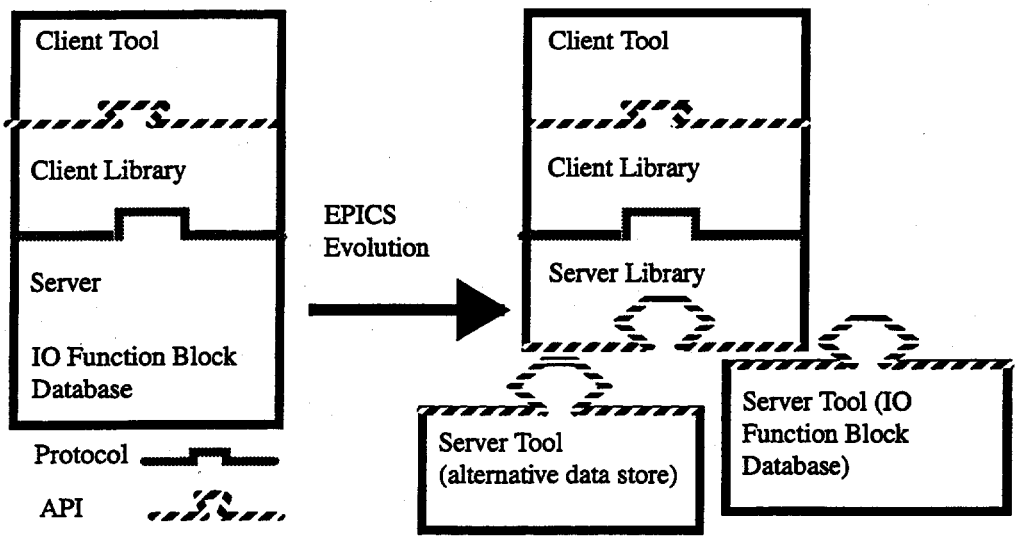
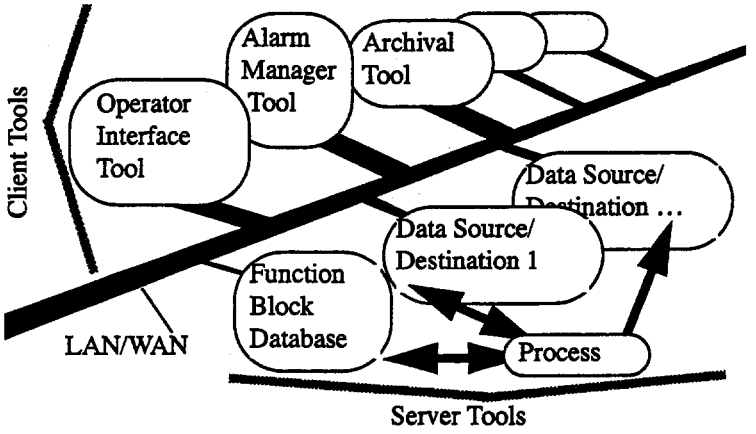


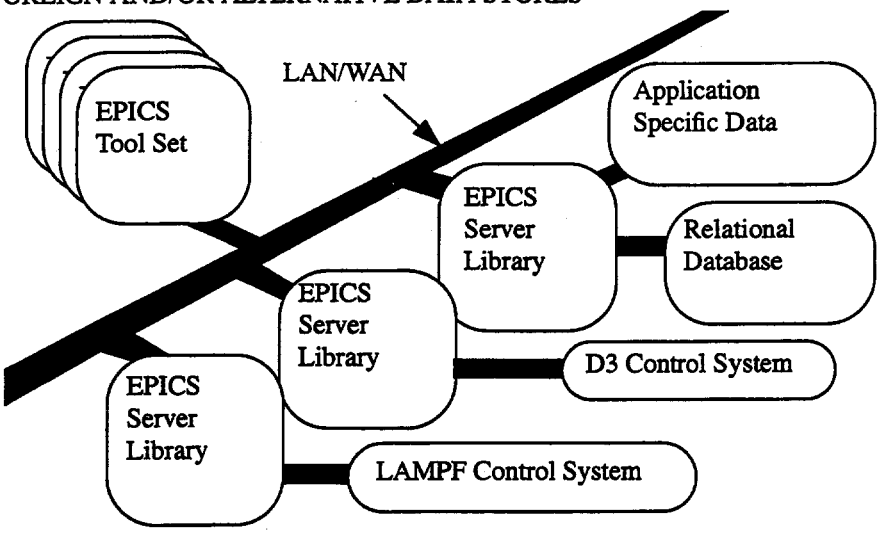
FIGURE 3. SERVER-SIDE API EXPANDS THE UTILITY OF THE EPICS TOOL SET



5.0 POTENTIAL APPLICATIONS

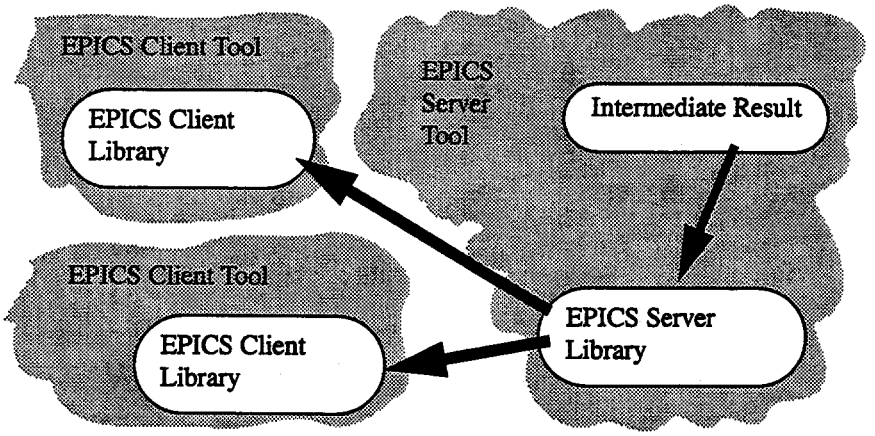
It is difficult to anticipate the full range of applications that might take advantage of the new interface. We list a few of the more important ones here in order to illustrate the practicality of this change. Perhaps the most important application will be to create EPICS server-side tools that will translate requests made by EPICS clients into actions made by foreign control systems (Figure 4). A prototype of this type of EPICS server (based on modifications to the original EPICS server source code) was demonstrated by Gabor Csuka at DESY and was used to interface between EPICS client-side tools and the D3 control system[8]. Next, the DESY-modified source code was further enhanced by Stuart Schaller at LANL for use as a gateway between EPICS client-side tools and the LAMPF/PSR control system[9]. We anticipate that the new server-level API for EPICS will become an integral part of similar projects. We predict a significant reduction in the labor required to implement and maintain gateways of this type because the new API is designed for this use and therefore cleanly packages server library functionality while carefully minimizing the source code that must be provided in-between the server library and the alternative data store (in the server-side tool).

FIGURE 4. FOREIGN AND/OR ALTERNATIVE DATA STORES



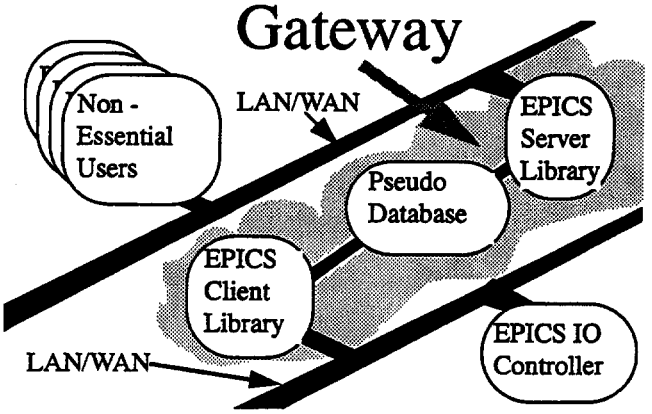
Another important application of the server-side API will occur when there is a tool that will receive certain parameters, perform some requested function, and then provide a result in the form of another parameter. In the past EPICS required that these intermediate parameters (between an EPICS client-side tool and some other tool) be stored in the IO function block database in the form of "soft" records. As stated above this isn't a natural place to store these parameters. The new server API allows *any* EPICS tool to export incoming and outgoing parameters (Figure 5). This approach allows new and existing applications to take advantage of the EPICS client-side tool set, with minimal effort, while avoiding the limitations described above. For example a modeling program might wish to take certain input from an operator interface tool and provide output back to the operator interface tool, an archival tool, and an IO controller. The new server-level API and associated libraries will allow the various elements of EPICS to be combined easily and then recombined in response to the changing requirements of a particular situation. The interconnection between tools will be determined by configuration and not by source code changes.

FIGURE 5. INTERMEDIATE RESULTS EXPORTED FROM A NATURAL LOCATION



Another important application of the server-side API is required when a large number of low-priority clients need access to an operational EPICS system. In this situation each client consumes system resources in one or more of the servers. If the number of low priority clients becomes too large there is the possibility that excessive loading might compromise critical functions performed by the server's processor. The new server-level API will make it possible to combine into one program the server and client libraries in order to create a gateway between low priority clients and critical machines. This gateway would serve as a proxy and guarantee that N clients consume no more resources on the critical machine than one client would consume by itself (Figure 6). Of course the penalty paid will be increased latency for low priority clients that are required to pass through a gateway. In return, clients that use the gateway receive less restricted access to the system because there is no longer any concern that they might overload it.

FIGURE 6. GATEWAY OFF-LOADS LOW PRIORITY CLIENTS



As a final example, the new server API will provide the necessary freedom to develop new front-end architectures for EPICS. Since it will be easy to install the server library into many different applications, there is the possibility of light-weight IO controller implementations which are dedicated to a particular task such as closed loop control. Since the design and development of these new architectures can proceed independently of the function block IO database source code maintenance we will be confident that we can add new features without risk of breaking existing installations. Since the new server library is being written to run on multiple platforms including UNIX, VxWorks,

MS windows and VMS, application designers will have the freedom to choose the operating system that best meets their needs.

6.0 INTERFACING DATA WITH THE CLIENT-SIDE TOOL SET

It is straightforward to interface data with the EPICS client-side tool set using the new server-level API. The application surrounding the data will become an EPICS server-level tool and the data will be exported as an EPICS process variable. A server-level tool must create a server instance and post change-of-state events when the data is modified. The server tool must also supply functions to be called when the process variable is located (existence test), attached to, detached from, read or written. A function that returns a valid range for the data must also be supplied. A server tool supplied function is also called when the client initiates or terminates monitoring the state of the data. The functions above define the minimum interface. Additional interfaces are required only if you wish to access a greater range of EPICS functionality. For instance, additional functions must be provided by a server-level tool if it needs finer control over a client's access rights when a client is modifying or reading the data.

7.0 CONCLUSION

With the new server-level API, projects will have greater freedom to choose and combine components of EPICS. The new interface will allow for modular development of new subsystems with minimal duplication of effort. The new interface is designed to encourage the development of tools for one project that may be used effectively on other projects and at other sites. A wide range of existing and future software systems will be able to efficiently take advantage of the EPICS tool set. The new API allows this to occur even when these system share little else architecturally with EPICS. Exporting data to the EPICS client-side tool set is straightforward when the new API is used.

[1] B. Kuiper, "Issues in Accelerator Controls", Proc. ICALEPCS, Tsukuba, Japan, 1991, pp 602-611.

[2] W. McDowell et al., "EPICS Home Page", "http://epics.aps.anl.gov/asd/controls/epics_home.html".

[3] L. Dalesio et al. "The Experimental Physics and Industrial Control System Architecture: Past, Present, and Future", Proc. ICALEPCS, Berlin, Germany, 1993, pp 179-184.

[4] L. Dalesio et al, "The Los Alamos Accelerator Control System Database: A Generic Instrumentation Interface", Proc. ICALEPCS, Vancouver, Canada, 1989, pp 405-407.

[5] J. Hill, "Channel Access: A Software Bus for the LAACS", Proc. ICALEPCS, Vancouver, Canada, 1989, pp 352-355.

[6] J. Hill, "EPICS Communication Loss Management", Proc. ICALEPCS, Berlin, Germany, 1993, pp 218-220.

[7] M. Knott et al, "EPICS: A Control System Software Co-development Success Story", Proc. ICALEPCS, Berlin, Germany, 1993, pp 486-491.

[8] M. Clausen, G. Csuka, Internal Communication.

[9] S. Schaller et al, "Generalized Control And Data Access At The LANSCE Accelerator Complex —Gateways, Migrators, And Other Servers", these proceedings.

Client-Server Design and Implementation Issues in the Accelerator Control System Environment *

S. Sathe, L. Hoff, T. Clifford
Brookhaven National Laboratory
Upton, NY, 11973-5000, USA

Abstract

In distributed system communication software design, the Client-Server model has been widely used. This paper addresses the design and implementation issues of such a model, particularly when used in Accelerator Control Systems. In designing the Client-Server model one needs to decide how the services will be defined for a server, what types of messages the server will respond to, which data formats will be used for the network transactions and how the server will be located by the client. Special consideration needs to be given to error handling both on the server and client side. Since the server is usually located on a machine other than the client, easy and informative server diagnostic capability is required. The higher level abstraction provided by the Client-Server model simplifies the application writing, but fine control over the network parameters is essential to provide the performance required. These design issues and implementation trade-offs are discussed in this paper.

1. Introduction

Very large scale integration and the advent of data communication networks have made desktop computers an affordable alternative to centralized facilities. Data-communication networks connect the computers together, allowing the exchange of information and the sharing of resources between different computers on the network. Resources can now be concentrated in the computer that best provides the resource and that computer can make the resource available to other computers via the network. An application is no longer confined to the resources available on the local computer, but can now use the resources available to the network.

An accelerator control application is an example of such a paradigm. It uses various services such as the database, accelerator device control, alarm handling, data archiving, data display and user interface services. To perform these services one or more of each type of server is available. These servers are distributed across the network and need to be accessible to the application. An application user should be able to access these services on the network without explicitly requesting the network transactions. The computer software should automatically locate the resource and transfer the information to and from the service. In other words, access to the services on the network needs to be transparent. A standard model for such distributed applications is a Client-Server model.

2. Client-Server Model

In a Client-Server model, the server offers the services to the network which the client can access. The term client and server do not necessarily imply computers; they can be thought of as a client process and a server process. In certain cases even a server process may perform a client's role in addition to its server role and vice versa. A Client Server relationship is not symmetrical[1]. This means that they are coded differently. The server is started first and never terminates unless it is forced to.

* Work performed under the auspices of the U.S.Department of Energy

A server typically opens a communication channel and waits for a client request to arrive at the well known address. Upon the arrival of a request, the server executes it in the context of the server process or in a separate one and sends back the results to the client. Then it goes back in the wait state to receive more client requests. The client, knowing the server address, opens a communication channel, connects to it, and then sends request messages to the server and receives the responses. When done, the client closes the communication channel.

A Client-Server model is considered to be part of the session layer and presentation layer of the well-known Open System Interconnect(OSI)[2] Model. This layer hides the application layer from some networking details and differences in data formats between various computer architectures. These higher level abstractions namely Client and Server provide an appropriate interface which makes the distributed application writing simpler.

3. Server Design

A server typically provides a number of services. A service is a piece of code that accomplishes the desired functionality. A Service can be fully defined by its name, input parameters and the results produced. Such a service can be executed in the context of the server process and is called an iterative service, or it can be executed in the context of another process and is called a concurrent service. The iterative services are used when the time to handle a request is known ahead of time. In the case of a concurrent service, the amount of time required to handle the service is unknown or is too long to hold the server process from accepting new requests. Concurrent services need to be reentrant and, if they have to share any global data, a proper locking mechanism is required. Accelerator controls device services such as setting the setpoint of a device or getting the readback from a device are examples of the concurrent type services, as the time required for these services varies with which control device is being used. However, the service that gets server diagnostic information can be of an iterative type of service.

A server can be *stateless* or *stateful*. A stateless server does not maintain any information or state about the clients. However a stateful server accumulates client information to function properly. In the case of a stateless server crash, the client comes to know about it and can retry to contact it. The server can just be restarted and then functions normally. However if a stateful server crashes in the middle of its operation, the server alone has the information to know where to resume operation. Server crash recovery can be complicated. A stateful server also needs to know about a client crash so that it can clean up the client information held with it. An accelerator controls device server that sends back a number of replies for a single client request needs to remember the client address and therefore is an example of a stateful server. However a display server is a stateless server since it does not have to remember any client information.

Another issue in server design is security. Should the server need to identify the client before accepting the request? If the server does employ some identification checking scheme, it should report security faults to some authority. Accelerator control facilities that give control system access to a large user community tend to have some kind of security scheme built in their system.

The issue of heterogeneity is important in the server design. Several kinds of heterogeneity need to be considered: machine architecture independence, operating system independence, software vendor implementation independence and server release independence. Different machine architectures have different data representations. Using higher level languages can solve this problem. The use of standards and portable compilers give the operating system independence. The server release independence implies that the client should be able to run independently of which version of the service is available. Vendor dependencies must be eliminated to increase the portability of the application.

Accelerator controls applications can be written using C or C++ languages to achieve the machine architecture independence. The use of a portable compiler such as GNU (provided by Open Software Foundation) C or C++ compiler gives operating system independence. The use of standard libraries such as POSIX gives vendor independence. In accelerator control applications it is common that a server and/or a client needs to be updated after it has been released. This need may be because of added functionality or a bug fix in the server code. It is often desirable that the old and new versions of server should coexist such that the new server can service the requests from the old or the new clients. The clients should be prepared to use the new server if it exists or should try the old one.

Error reporting is one of the important features of the server. A server needs to return the good or bad status of the service executed. A well defined interface to define all the service-related errors is crucial.

4. Client Design

A client is an entity that requests services from a remote or a local server. The client assembles a request message and transmits it to the server to initiate some action by the server. The first step in a client design is to determine how the client will find the server process to which it wants to send the requests. Some kind of a database is usually employed to hold this information.

The request messages sent by a client to a server can be broadly categorized as send-only, blocked, callback, batch and broadcast[1]. A send-only type message originates at the client end and is sent to the server. There is no reply expected from the server for this message. A client request sent to the display server to update the data to be displayed is an example of a send-only type message. When a blocked message is sent to the server, the client blocks until the reply is received from the server. A request to get the control device server diagnostic information is an example of such a type of message. When a callback message is sent to the server, one or several replies are expected from the server at a later time. To receive such delayed replies, the client now has to become a server and the server has to become a client while originating the replies. For example, an accelerator control device client sends a callback request to a server to receive the data from a device based on a hardware or a software event. A broadcast message is sent to probe the network for servers matching a certain address. The servers matching this address acknowledge the request by sending a reply back to the client. A batch message keeps the requests at the client side until the client lets them go over the network. An advantage of sending requests in batches is that it reduces the network overhead. A single reply for all the requests is sent by the server. Accelerator control clients use batching of request messages to improve overall performance.

There are some issues to consider while determining the timeout values for the client. Servers are likely to take varying amounts of time to service individual requests, depending on factors such as server load, network routing and network congestion. The client should be prepared for the worst conditions or for a variation of service time-outs.

A client can fail to communicate to a server for various reasons. For example, the client may not find the address of the server, or the network between the server and client may not be operational, or the machine on which the server runs may not be up, or the server itself may not be running. The client needs to detect and report these errors in a well-defined fashion.

A client and server running on two computers having different architectures pose a data interpretation problem. To overcome such a problem various strategies can be used. The client can filter the data into a machine-independent format before sending it to the server. The server on receiving the request filters it in its native format. When sending the reply back to the client, the server filters the data in the machine independent format and the client filters it back into the native format.

A second strategy could be that the server always makes the data right after receiving and before sending. This strategy assumes that the server knows about its native architecture data formats as well as the client's architecture data format. Another strategy is that the client always makes the data conversions before sending and after receiving. In this case the client has to know about its native as well as servers's architecture data format. It is also possible to have the receiver always making the data right. In such a case both client and server have to know the architecture of the machine from which the data came. Accelerator control applications can choose from one of the above mentioned techniques that is suitable for their environment. However the technique that converts the data to machine independent format or the case where the receiver always makes the data right are supported by standard industry tools such as RPC[1][3].

5. Client Server Performance

As with any software design, performance is an issue in the design of the server. Numerous client requests can quickly affect a servers's performance, if the server has to do a lot of processing for each request. By keeping the request short and the amount of work required by the server for each request low, the performance can be improved, especially in the case of the iterative server. If the service takes a long time to finish, the server performance can be improved by making it concurrent. If the concurrent service uses a globally shared resource, care should be taken to lock it at the lowest possible level of granularity to avoid delays and assure smooth working of the server.

A client should try to group small requests into one batch and then send it to the server in one network transaction to avoid the overhead involved in sending individual small requests.

One of the parameters that has a big impact on the server performance is flow control. Flow control assures that the client does not overwhelm the server by sending requests at a faster rate than the server can process them. The size of the request message and the rate at which the message is sent need to be tuned for the given network configuration.

Proper network parameter selection is important both on the client and the server side. In the accelerator control applications, the message size typically varies from application to application. It ranges from a few bytes to a few hundred kilobytes. The time required to send and receive the message is mainly dependent on the size of the message for the same distance. It is desirable to be able to set the timeout suitable for a given request. The network receive buffer size for the server is a function of the largest message size, as well as how many clients are expected to communicate to the server simultaneously. The network send buffer size needs to be set as well, depending upon the size of the message and the rate at which they are sent. To help the user to get a handle on the network transaction timing, the client needs to provide the timing statistics for the messages being sent and the reply messages being received.

Last but not least, the network components play an important role in improving the client server performance. High performance network elements such as bridges and routers and high band-width networks, specially for consoles that collect data from a number of front ends, are crucial.

A server health checking mechanism is necessary to be built in the server design. Some diagnostics about the server request handling are highly desirable.

6. Client Server Implementation

One of the major decisions that the implementor needs to make in the beginning is what network transport is appropriate for a given Client Server model. User Datagram Protocol (UDP) and Transmission Controls Protocol (TCP/IP) are widely used transports in accelerator control system. The size of the messages to be exchanged, network topology and reliability of the message delivery are important determining factors amongst many others. UDP seems to be suitable for smaller size messages, typically less than 1000 bytes and for the smaller network. The smaller message size and smaller network ensure a minimal packet loss with normal network traffic. TCP/IP is desirable in case of large message sizes and for the wider networks. It provides a reliable data delivery and also does the flow control so that the sender does not overload the receiver by sending data at a rate faster than it can handle. TCP/IP being a connection oriented protocol, the client needs to reconnect after a server crash.

Having selected the transport, one proceeds to choose the interface to be used to implement the Client Server Model. Remote Procedure Call(RPC) is a well known mechanism that is used to invoke a procedure on a remote system. The RPCs prevent the client and servers from having to worry about details such as sockets, network byte order etc. which makes distributed application writing easier. Some accelerator control system designers choose to write their own RPCs while others utilize the standard ones. Standard RPCs enable the writing of servers and clients in a uniform way. Typically, they provide standard ways for finding the server process on a given host. The standard RPCs provide a mechanism to define the request and reply messages which is vital to any distributed application. Each type of message can be defined by its name. The request and reply data also can be defined in terms of single data items or an

arbitrary structure. Errors are handled and reported via a well defined interface. Security mechanisms, both on the server and the client side are provided by the RPC interface. Since RPCs are available on various Unix as well as Real Time systems, the client server code becomes portable. Various machine architecture heterogeneity is taken care of by the standard RPCs. They also provide a uniform health checking mechanism crucial to any distributed application. RPCs provide a mechanism to structure the request and reply data in an arbitrary, user defined fashion. RPCs in general are well suited for synchronous type of communication, where the client blocks until the reply from the server is received. To implement the callback type of message delivery, which is asynchronous in nature, takes extra efforts on the part of the implementor.

Using the concepts described above, a Client-Server model has been designed and implemented for the AGS and RHIC control systems. There are two different implementations, one for each control system, because of different requirements and historic reasons. UDP transport was found suitable for AGS, because of the message size of 512 bytes and a small network of about 40 front ends. As UDP does not support the flow control, the clients needed to introduce the flow control explicitly. As the RHIC supports large message sizes and is planned to have of the order of 150 front ends, TCP/IP was a natural choice. Both the AGS and RHIC accelerator device servers are designed to be stateful. Since TCP is a connection-oriented protocol, the design needed to provide mechanisms for cleaning up the client information from the server as the clients crash. Both iterative and concurrent services are supported by the servers. As the vendor supplied software does not give a handle on the client-server connection timeout, a Unix signal is used to interrupt the system connect call. In the case of a server crash, the TCP-based clients need to reestablish the connection with the server. In contrast, UDP based clients do not have to worry about it. Both blocked and callback type client messages are supported. Client-server implementation is a C++ class library and is portable across Unix and VxWorks operating systems. The class library is based on the standard SUN Open Network Computing(ONC) RPC communication interface. The capability of adjusting the network buffer size and time-outs is also provided. The rpcinfo program supplied by RPC is used for checking the health of the server. To get a handle on more server specific information, the server diagnostics provides information such as start-up time, the machine name on which it is running, the number of synchronous and asynchronous messages it has handled from the start-up time and so on. It also provides the information about callback clients. Typical diagnostic information is as follows:

ADOIF SERVER DIAGNOSTICS INFO

Host Name: acnfec007.rhic.bnl.gov
startupTime: THU OCT 19 08:30:28 1995

RPC Program Number: 1000002
RPC Version Number: 0
TCP Socket Number: 19
Port Number: 990
Receive Queue Size: 10000 bytes
Send Queue Size: 10000 bytes

Synchronous Messages handled: 2784
Asynchronous Messages sent out: 9178
Asynchronous Active Requests: 328
Async Clients Being Served: 4

Async Client Addresses being used by the ADOIF Server

Client Address No 0

Host Name: acnindy04.rhic.bnl.gov
RPC Program Number: 1073742096
RPC Version Number: 1

Server Port Number: 10998
Process Id: 16272

Client Address No 1

Host Name: acnindy02.rhic.bnl.gov
RPC Program Number: 1073742226
RPC Version Number: 1
Server Port Number: 12741
Process Id: 1402

Client Address No 2

Host Name: acnindy02.rhic.bnl.gov
RPC Program Number: 1073742231
RPC Version Number: 1
Server Port Number: 12744
Process Id: 1407

Client Address No 3

Host Name: acnsun17.pbn.bnl.gov
RPC Program Number: 1073741829
RPC Version Number: 1
Server Port Number: 44807
Process Id: 26005

7. Conclusions

The Client-Server Model is a standard model used in the accelerator controls applications. There are various server and client design issues. They include concurrent versus iterative services, stateless versus stateful servers, message security, machine architecture, software vendor and server version independence. The design of built-in mechanisms to send and receive different types of messages such as send-only, blocked, callback, broadcast etc. is necessary. To improve the server performance, proper flow control on the client side is necessary. Selection of network time-outs and selection of proper network buffer sizes is a key to performance tuning. Standard RPCs are well suited to implement a Client-Server model as it addresses most of the design and implementation issues of such a model. A Client-Server model implementation that handles callback type messages tends to be more complex and involved than one that handles only synchronous type messages.

References

[1] J. R. Corbin, The Art of Distributed Applications, Springer-Verlag New York
[2] W. Richard Stevens, Unix Network Programming, Prentice Hall
[3] W. Rosenberry, D. Kenney, G. Fisher, Understanding DCE, O'Reilly Associates, Sebastopol, CA
[4] J. Bloomer, Power Programming with RPC, O'Reilly Associates, Sebastopol, CA

FERMILAB-Conf-96/069

Automatic Generation of Configuration Files for a Distributed Control System

J.Cuperus, A.Gagnaire

PS/CO Group, CERN, CH-1211, Geneva23, Switzerland

ABSTRACT

The CERN PS accelerator complex is composed of 9 interlinked accelerators for production and acceleration of various kinds of particles. The hardware is controlled through CAMAC, VME, G64, and GPIB modules, which in turn are controlled by more than 100 microprocessors in VME crates. To produce startup files for all these microprocessors, with the correct drivers, programs and parameters in each of them, is quite a challenge. The problem is solved by generating the startup files automatically from the description of the control system in a relational database. The generation process detects inconsistencies and incomplete information. Included in the startup files are data which are formally comments, but can be interpreted for run-time checking of interface modules and program activity.

INTRODUCTION

The block diagram of the CERN/PS accelerator control system is represented in fig 1. The Control Modules [1] are software modules which present a uniform call interface to the users. These Control Modules address the Equipment Interface, to which the accelerator equipment is connected. The Equipment Interface is composed of a software part: programs and interface module drivers, and a hardware part: crates with plug-in modules. The Equipment Interface is controlled by Device Stub Controllers (DSCs), which are microprocessors sitting on a VME board.

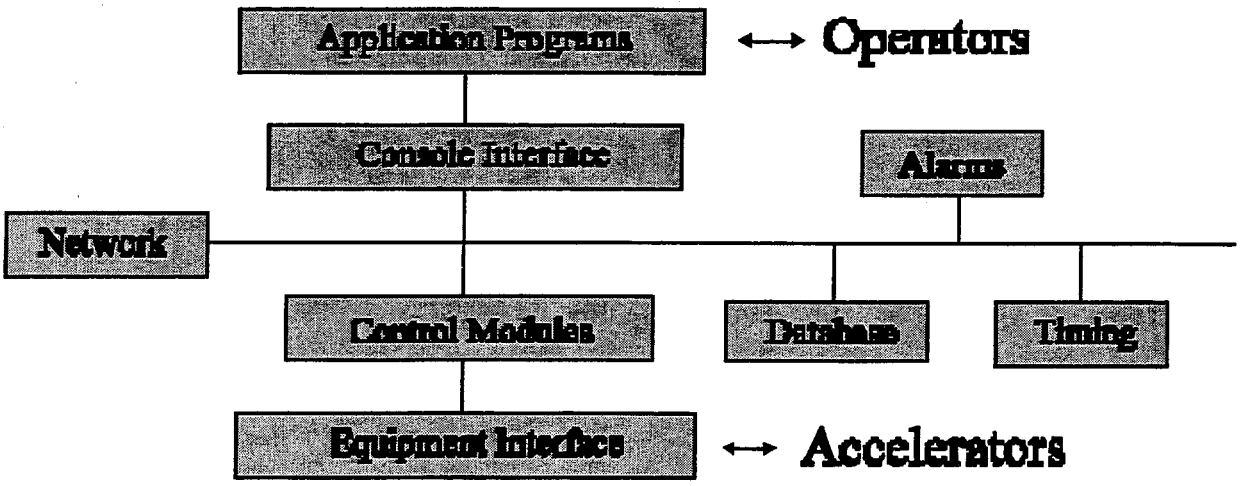


fig 1: Block diagram of the control system. What interests us here is the Equipment Interface, which is composed of programs, drivers, and hardware modules in VME, CAMAC, G64, and GPIB crates.

The Problem

The LynxOS operation system (a real-time UNIX) expects to find a rc.local initialization file for the DSC, with information about programs to be started, drivers to be installed, addresses, and interrupt vectors. Initialization files are typically 100 lines long and filling them in by hand is quite a problem:

- We have now more than 100 DSCs and the management by hand of each rc.local file is quite difficult and a waste of time.
- The startup sequence is made of related pieces and any change can have perverse side effects and lead to a fatal system fault or misbehavior of application programs.
- There is no validity check for conflicts in address ranges or interrupt vectors and the declarations are not checked against the really installed hardware.

The Solution

In fact, we had most of the data already in our relational database and, with a few extensions, we can now get all data from tables in the database. Filling in these data, with the help of forms, is easy and the tables are organized in such a way that duplication of information is avoided . Most parameters have default values and must be filled in only when a different value is required, which is exceptional.

We decided to use this information to generate the rc.local files automatically with a data driven program. This program performs several checks on the data and informs the user in case of conflicts. We will now look at this process in some more detail.

HARDWARE DESCRIPTION

Accelerator Interface Configuration

The hardware interface is connected to the rest of the control system through a DSC, which is a front-end microcomputer sitting on a VME board. The CPU is a Motorola MVME167 microprocessor. On one side it is connected to the Ethernet network and on the other side to the VME bus. All hardware is controlled through the DSC, either directly though modules (cards) on the VME bus or, indirectly, through VME driver modules for CAMAC, MIL1553, or HPIB loops (fig. 2).

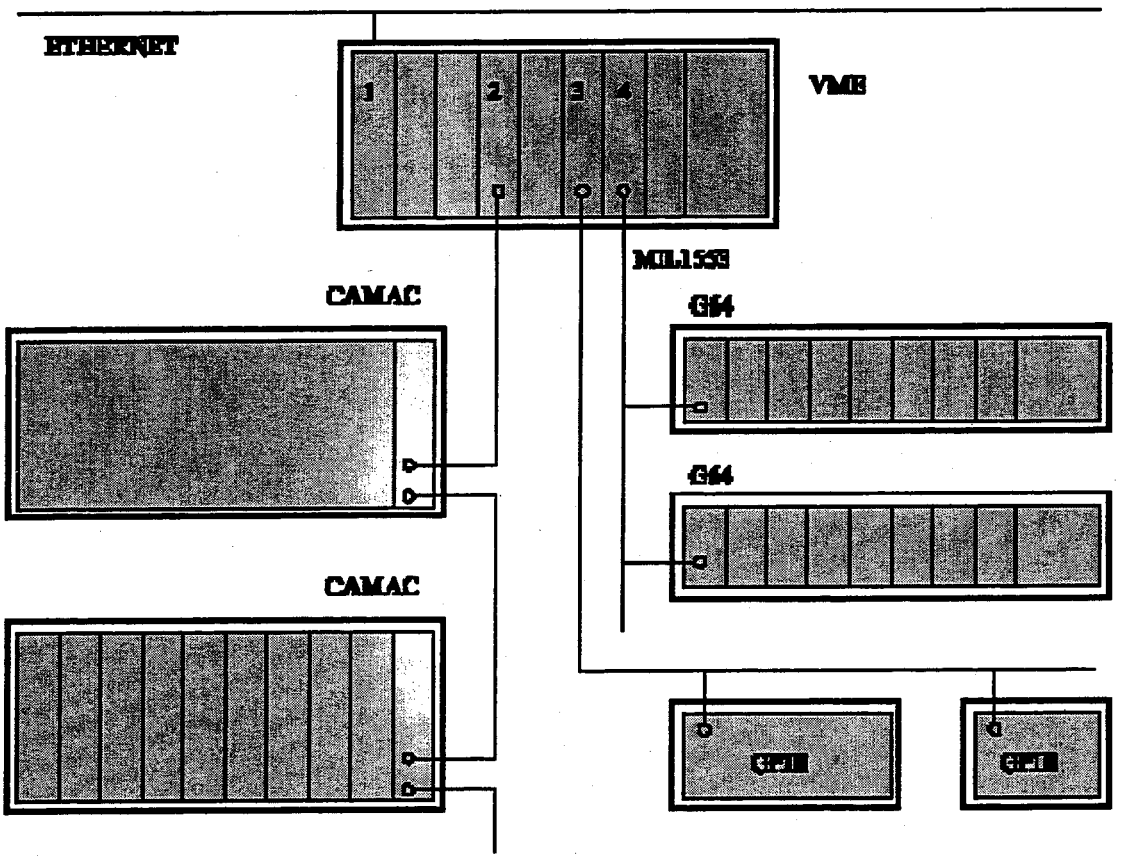


fig. 2: Accelerator interface configuration: (1) DSC microcomputer, (2) Serial CAMAC driver, (3) General Purpose Instrument Bus driver, (4) MIL1553 driver.

On these loops (or busses) can sit CAMAC, G64, or GPIB crates with slots for modules. These modules are, in general, connected to the accelerator hardware: power-supplies, instruments, actuators, etc...

Database Tables describing the Hardware

The whole interface configuration is described in tables in the Oracle relational database management system. We can distinguish two groups of tables:

- *TYPE* description tables, which contain a description of the object type or class. Tables COMPTYPES, CRATETYPES, and MODULETYPES contain all the fixed attributes per type of computer, crate or module.
- *INSTANCE* tables which describe the implementation of an instance of the type. The installed computers, crates, and modules are instances of types but they have attributes of their own, which are entered in following tables:

COMPUTERS = { CompName + CompType + . . . }
CRATES = { Crate_Id + CrateType + CompName + LoopNo + CrateNo + . . . }
MODULES = { Mod_Id + ModuleType + Crate_Id + Slotno + Seqno + . . . }

The type of computer that interests us here, is the DSC. A crate is any chassis with slots of type VME, CAMAC, G64, or GPIB. A module sits in a slot of a crate. Additional details about module instances can be found in following tables:

MOD_EXCEPTIONS = { Mod_Id + DriverName + InterruptLevel + BaseAddress1 + . . . }
MOD_INTERRUPTS = { Mod_Id + InterruptNo + Subaddress }

The MOD_EXCEPTIONS table is separate from the MODULES table because modules need an entry only when the values calculated by default are not suitable, which is quite exceptional. The MOD_INTERRUPTS table is separate from table MODULES because a module can generate several interrupts.

Finally, there are the tables EQUIPMENT and INSTVAL, which are related to the Control Modules and which contain, among other things, physical addresses for VME, CAMAC, G64, or GPIB hardware modules, for controlling pieces of accelerator hardware.

SOFTWARE DESCRIPTION

Software Modules

Software modules come essentially in two kinds:

- Hardware drivers which hide some of the intricacies of the hardware control for a type of hardware module.
- Programs which provide some control or surveillance function.

Some of the external aspects of software modules can be described in Oracle tables.

Database Tables for describing the Software

We can again make a distinction between type description tables and instance description tables. The different driver types are described in following table, with default parameters and expected tags for startup sequences:

DRIVERTYPES = { DriverName + DriversPrio + Address_Tag + . . . }
DSCPROGDEFS = { StatupName + Progname1 + Progname2 + StartSequence + . . . }

The StartSequence can contain placeholders $1..$4, which can be replaced by parameters in table DSCPROGRAMS. This table contains the list of programs to be started in each computer, and drivers to be installed:

DSCPROGRAMS = { CompName + Seqno + ProgName + Prio + Params + . . . }

DATA ENTRY WITH FORMS

All data are entered into the database through Forms, of which fig. 3 gives an example. All forms for this application, together with provisions for starting all necessary programs and actions are grouped in a menu

structure. This menu structure is the only interface needed for entering data and generating the needed configuration files. After filling in the data and choosing a DSC from a list, you can select the "generate configuration file" item from the menu.

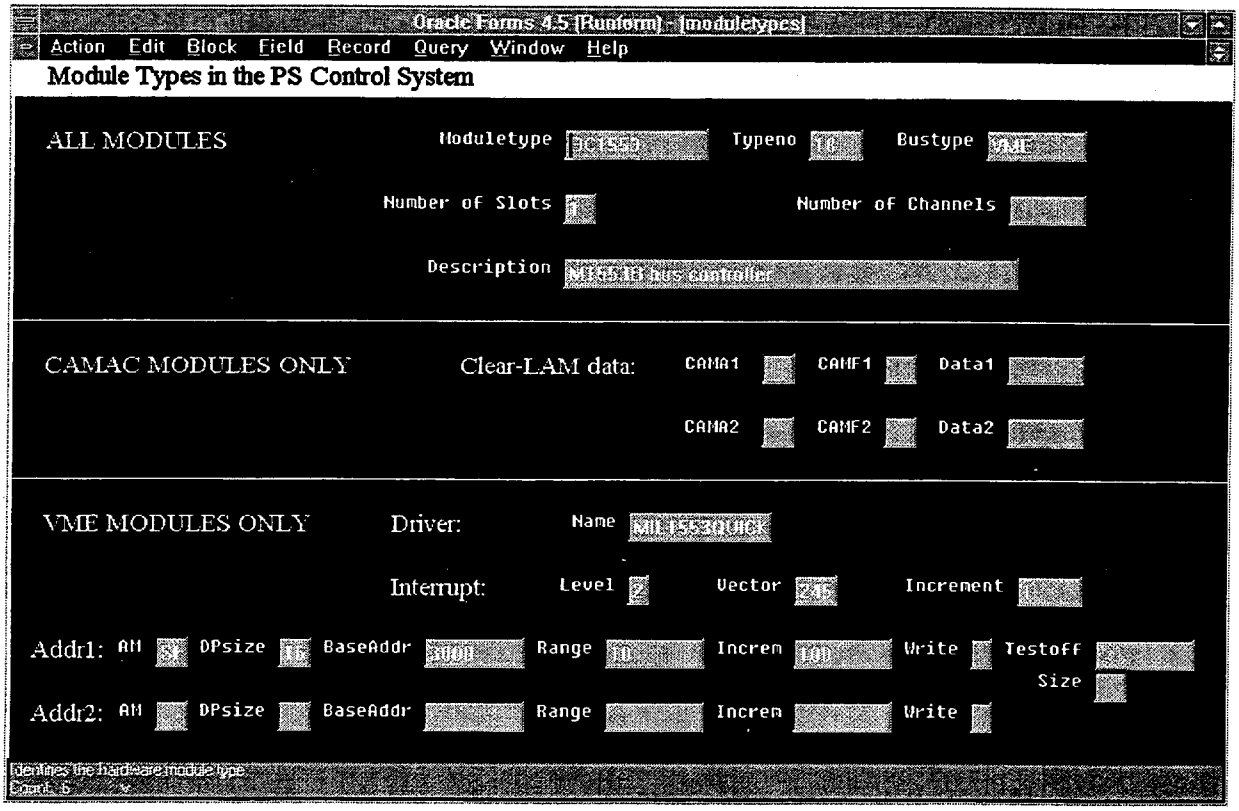


fig. 3 A data entry form

FILE GENERATION

Default Address and Interrupt Vector Calculation

Every VME module type has one or two base addresses and an interrupt vector. The first module of this type installed in a crate (with Logical Unit Number - or LUN - equal to 0) adopts these values as defaults. Subsequent modules of the same type installed in this crate get LUNs 1, 2 , 3 ... and the following default values:

ADDRESS = BASEADDRESS + LUN*ADDRESS_INCREMENT

INTERRUPT_VECTOR = BASE_VECTOR + LUN*VECTOR_INCREMENT

If none of the address ranges or interrupt vectors in the crate overlap, then all is OK. If not, different values must be entered in the table MOD_EXCEPTIONS. The need for this is indeed exceptional, if the type defaults have been selected with care.

Configuration File Generation

The generation program asks for the name of the DSC and then, with the help of the data in the database, executes a number of checks:

- **Check VME, CAMAC, G64, and GPIB:** all equipment addresses must correspond to an installed module.
- **Check crate slots:** physical space must be available in the crates for all modules.
- **Check interrupts:** calculate interrupt vectors and check for conflicts.

- **Check base addresses:** calculate base addresses and check for overlapping ranges.
- **Check Drivers:** required drivers must exist and work within their limitations.

If all is OK, then the program generates the following entries in a rc.local file:

- **Write IOCONFIG data:** comments which describe the VME modules and crates in the interface. These are not required by LynxOS but are readable by a configuration display program. They are also used, at startup, by a program which checks that all listed hardware is installed and addressable.
- **Write first set of program startup data:** programs which must be started before the drivers.
- **Write driver installation data:** install all drivers, in correct sequence.
- **Write second set of program startup data:** programs which must be started after the driver.
- **Write CLIC data:** comments readable by a program execution monitor, named CLIC which alerts the operator when one of the programs stops functioning.

An Example of a rc.local File

```
#!/client/dlsh
#1995-SEP-29
# dmcrsync startup file rc.local, generated 1995-SEP-29/11:14.
$#set(path, . /client /dsc/local/bin /dsc/bin/bin /dsc/bin/rt /bin )

#*****************************************************************************
#  WARNING :  File generated from database.
#          Can be overwritten at any time !
#
#  Latest dsc modifications made by : nmn_svps13_19Apr95_12:23
#*****************************************************************************

# ***** IOCONFIG Information *****

#   ln  mln     module-type       lu  W  AM    DPsz   basaddr1   range1  W  AM    DPsz   basaddr2   range2  testoff
#+# 1   0       VME SAC           0   N  SH    DP16   0          20      N  ST    DP16   0          80000   -
#+# 2   0       VME SDVME         0   N  SH    DP16   f800       400     N  --    ----       0          0       -
#+# 3   0       VME SDVME         1   N  SH    DP16   f000       400     N  --    ----       0          0       -
#+# 4   0       VME SDVME         2   N  SH    DP16   b800       400     N  --    ----       0          0       -
#+# 5   0       VME MVME147S      0   N  --    DP16   0          0       N  --    ----       0          0       -
#+# 6   0       VME PLS-REC-FPI   0   N  SH    DP16   e000       1000    N  --    ----       0          0       -
#+# 7   0       VME PLS-REC-FPI   1   N  SH    DP16   d000       1000    N  --    ----       0          0       -
#+# 8   0       VME PLS-REC-FPI   2   N  SH    DP16   c000       1000    N  --    ----       0          0       -
#+# 9   0       VME ICV196        0   Y  ST    DP16   500000     100     N  --    ----   0           0       -

#  ln   sln     module-type       lu  evno    subaddr    A1 F1 D1     A2 F2 D2

#   ln  mln     module-type       lp  cr
#+# 10  2       CAM SCC-L2        1   11
#+# 11  2       CAM SCC-L2        1   12
.
.
#+# 24  4       CAM SCC-L2        3   8
#+# 25  4       CAM SCC-L2        3   12

# ***** Program Startup before drivers *****

# ***** Driver Initialisation *****

-( cd /dsc/bin/drivers/sacvme;  sacvmeinstall \
 -R0 -M0 -V254 -L2  )-

-( cd /dsc/bin/drivers/sdvme;  sdvme_v2install \
 -Af800 -V160 -L2  -LP1  -, \
 -Af000 -V161 -L2  -LP2  -, \
 -Ab800 -V162 -L2  -LP3  )-

-( cd /dsc/bin/drivers/fpiplsvme;  fpiplsvmeinstall \
 -BOe000 -BV172 -BL2  \
 -AOd000 -AV173 -AL2  \
 -COc000 -CV174 -CL2  )-

-( cd /dsc/bin/drivers/icv196vme;  icv196vmeinstall \
 -AO500000 -AV130 -AL2  )-

# ***** Program Startup after drivers *****

# Install data used by ioconfig library
-(cd /dsc/bin/drivers/ioconfig/; ioconfigInstall  )-

# Start errlogd reporting errors to mcrsrv
prio 16  errlogd mcrsrv </dev/null &
```

```
# Report messages from CLIC or IOCONFIGDIAG to mcrsrv
rm -f /dev/sysReport.pipe
prio 17 sysReporter mcrsrv &

# Survey of VME and CAMAC loops according to configuration
prio  10 /dsc/bin/bin/ioconfigDiag  </dev/null &

echo Restoring datatable and run Nodal sys_go
dtrest
if $access(r,all,/dsc/bin/rt/sys_go.nod)
nodal ``$se ex.fl="";lo /dsc/bin/rt/sys_go.nod;run"</dev/null
end

prio 21   global_event_server \
  /dsc/local/bin/global_event_configuration </dev/null &

echo start NODAL EXEC server
prio 19   usr_cmd mcrop `nodal  -s EXEC'

echo start NODAL IMEX server
prio 19   usr_cmd mcrop `nodal  -s IMEX'

echo start equipment RPC server
prio 19   server </dev/null &

# echo start CLIC alarm survey
# $#setenv(USER,root)
# rm -f /dsc/local/clic/.clic.lockdaemon
# rm -f /dsc/local/clic/.clic.lockrun
# prio 10 clic -s

# ***** Programs in Clic Survey *****

#-# errlogd
#-# server

# End of file rc.local : all OK.
```

CONCLUSIONS

The system is a great help in configuring our many DSCs. Instead of laboriously filling in the configuration files by hand, which is a very error prone process, we derive them from descriptions in the database. The database is easier to fill and minimizes the entry of repetitive information and the data are necessary for other purposes anyway. As a bonus, the hardware configuration is checked at startup and programs are surveyed at runtime. The descriptions are broad enough to cover all special cases and we never find it necessary to edit the configuration files by hand.

REFERENCES

1. L.Casalegno, J.Cuperus and C.H.Sicard, Int. Conf. on Accelerator and large Exp. Physics Control Systems, Vancouver, 1989, D.P. Gurd, M.Crowley-Milling, eds. (North-Holland, Amsterdam) p 412.

2. More information about the PS accelerator control system can be found on the web address: http://psas01.cern.ch/

THE TECHNICAL DATA SERVER FOR THE CONTROL OF 100 000 POINTS OF THE TECHNICAL INFRASTRUCTURE AT CERN

P. Ninin, H. Laeger, S. Lechner, R. Martini, D. Sarjantson, P. Sollander and A. Swift
CERN, Geneva, Switzerland

Abstract

We have defined a Technical Data Server (TDS) to be used for the supervision and control of the technical infrastructure of CERN by its dedicated control room, the Technical Control Room (TCR), and by equipment groups. The TDS is basically a real-time information system which contains all states of the technical infrastructure: 100 000 points describing electrical distribution, cooling water, air-conditioning, vacuum, safety, and similar systems. It is expected that the TDS will substantially increase system performance and ease operation. As the concept of such a data server is also of interest to other groups in CERN accelerator and experimental physics divisions, detailed software and user requirements, as well as criteria for selection, implementation and maintenance have been elaborated in common with these groups. The project adheres fully to the European Space Agency (ESA) PSS-05 Software Engineering Standards and its life-cycle approach. This paper describes the data acquisition and distribution mechanisms, the interfaces to equipment and to existing alarm and data logging systems and to operator supervision consoles, the alarm reduction mechanism, and the error-handling and logging. Problems encountered during the project development are discussed in some detail.

I. INTRODUCTION

At CERN, the Technical Control Room (TCR) monitors data coming from the electrical distribution, cooling water, air conditioning, vacuum, cryogenics, safety, and other systems. In this context, a Technical Data Server (TDS) has been defined and will provide data collected from the above equipment to high-level control software such as Human–Computer Interfaces (HCIs), the logging system and the alarm server. An expert system will be used to perform alarm filtering.

This project is being developed using the European Space Agency (ESA) PSS-05 Software Engineering Standards [1].

The intention of this paper is to highlight the scope of the TDS project, the project environment and how we intend to achieve the project goals (methods, tools). There will also be a detailed description of the chosen middleware package, RTworks by Talarian.

II. MOTIVATION FOR THE PROJECT

The project involves a redesign of the existing methods of acquiring and managing data used in the TCR. At present, data for the HCI and logging is polled from the equipment directly (top-down) and alarm data is handled by event-driven software (bottom-up). Data is acquired through a large distributed network that covers all accelerator sites: PS, SPS, LEP, Meyrin and Prévessin.

The benefits of the TDS are numerous. It keeps a permanent image of all equipment attributes monitored by the TCR (100 000 points). It proposes a new, more reliable channel for alarm transmission to cope with alarm bursts and a centralized alarm reduction mechanism. The response time of the Uniform Man–Machine Interfaces (UMMIs: applications which display equipment states), is improved, to respond to data requests within one second. Equipment access is rationalized, as specific access routines are replaced by a generic addressing mechanism which allows both data retrieval and sending of commands. The use of an industrial middleware package decreases the maintenance effort for multiple client-server applications. The number of processes in each hardware element is decreased, due to the fact that multiple individual equipment access no longer takes place. Finally, the TDS offers a solution to the problem of supplying data to the increasing number of TCR applications users. The TDS provides high availability and reliability (24 hours a day, 365 days a year) as required for the operation of CERN services. All data monitored by the system are defined in a unique reference database and identified by a unique tag [2].

III. THE ENVIRONMENT

The environment consists of a three-layer architecture [3]: the Control Room layer, the Front-End Computing layer, and the Equipment Control layer (Fig. 1). The Control Room layer consists of HPUX servers (HCI, alarms, RDBMS) and X-Terminals. The HCIs are based on an XWindow–OSF/Motif user-interface management system and Dataviews [4]. The Front-End Computing layer consists of Front-End process computers (FEs), also called Process Control Assembly (PCA), based on PCs and VMEbus crates. They interface with various fieldbuses and more particularly the MIL-1553 fieldbus.

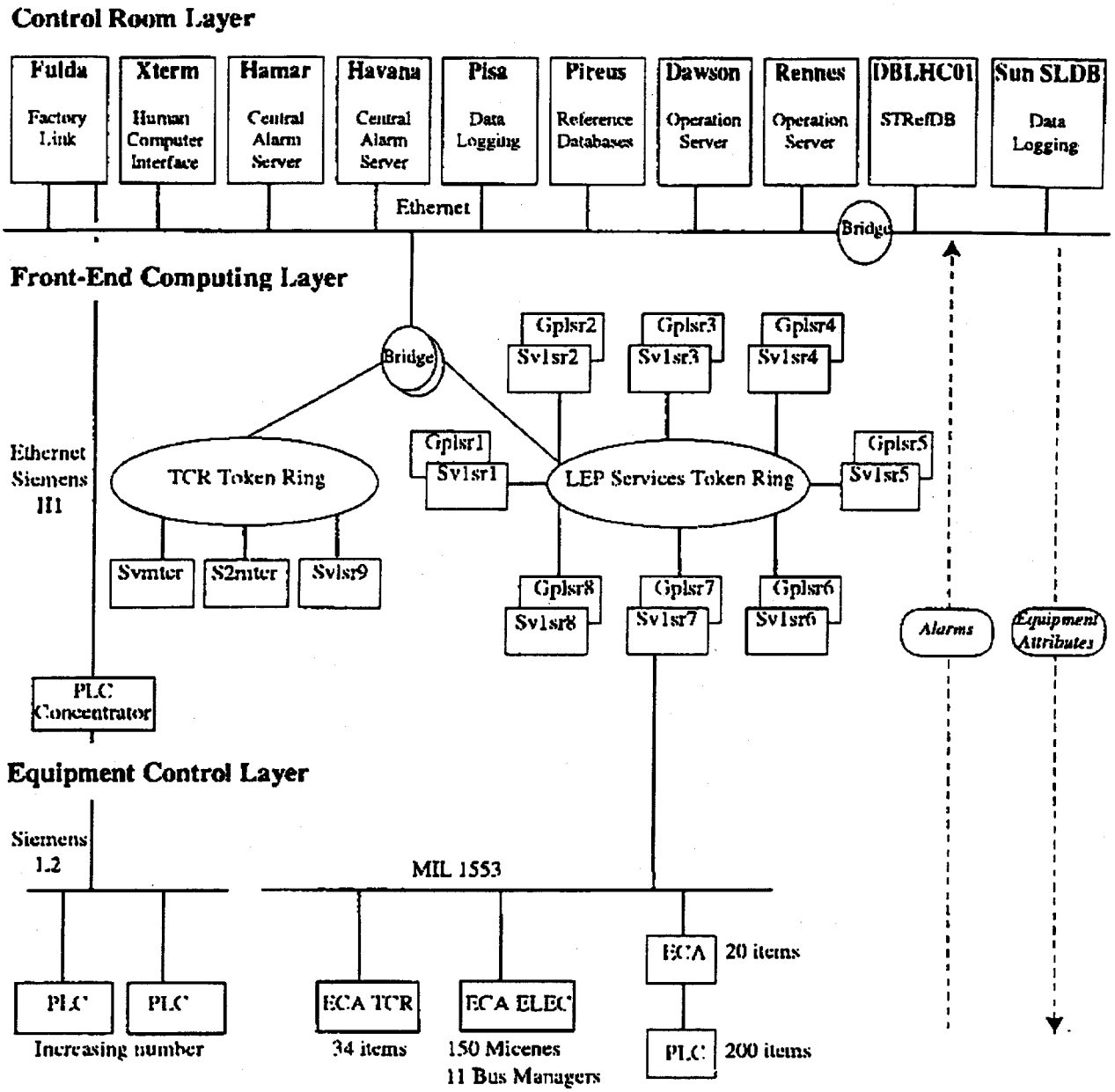


Figure. 1. The TCR Control Environment

The communication between the applications used in the Control Room layer and the Front-End layer is achieved through Remote Procedure Calls. The new generation of FEs consists of a standard VME rack powered by a PowerPC processor and running LynxOS. The Equipment Control layer consists of Equipment Control Assemblies (ECAs) connected to the FEs via various equipment fieldbuses (MIL-1553, GPIB, BITBUS, JBUS) or via RS232/422 links. At some places the ECAs provide the interface to local control systems, such as Landys & Gyr or to programmable logic controllers (PLC).

Network communication is provided by local Ethernet segments bridged to large token-rings. There is one specific token-ring network for the LEP services. These networks will be replaced by a 100-Mbit/s FDDI backbone that will cover the entire CERN site.

The hardware environment for the target system will consist mainly of two types of machine. The first is the HP700 series which will execute the archive logging, logical analysis and data distribution for the information system. The second platform is the VME-bus PowerPC's 603/604 which will execute the data access module.

It is expected that the TDS will run under the HP-UX UNIX operating system, XWindows, X11 and LynxOS. All XWindows application development will adhere to the OSF/Motif standard.

IV. ACHIEVING THE GOAL

In order to achieve the proposed goal, the TDS project team adopted the ESA PSS-05 Software Engineering Standard [1]. After following the Standard through its first phase, the User Requirements phase, and into the second phase, the Software Requirements phase, a detailed logical model was created and a market survey was performed.

The market survey, which considered many middleware products, was conducted in order to determine whether the project could be implemented using a commercially available product. The investigation resulted in a comprehensive document that detailed each of the middleware packages most likely to satisfy the logical model. One particular package, RTworks by Talarian, was highlighted as being the product that would meet our needs.

V. MIDDLEWARE

A. Technical overview

RTworks is a suite of software development tools for building time-critical monitoring and control system applications. It consists of separate processes for data acquisition, data distribution, real-time logical analysis and graphical user interfaces. RTworks is specially designed for building applications where large quantities of data must be acquired, analysed, distributed, and displayed in real time. The RTworks processes communicate via a dedicated message server. Applications can run on a single workstation or can be distributed across multiple processors in a heterogeneous network.

The RTworks client-server architecture is built specifically to offer high-speed inter-process communication, scalabilty, reliability, and fault tolerance. As the needs of the application grow RTworks, user-development, or third-party software processes can be added transparently.

The major RTworks software processes are:

RTserver - information distribution server
RTie - expert system builder and inference engine
RThci - dynamic graphical user interface builder
RTdaq - data acquisition interface
RTarchive - intelligent information archiver
RTplayback - information playback module.

Tags describing similar equipment are gathered in 'datagroups'. Client applications subscribe to these datagroups and receive all related tag changes.

The design of RTworks fits very nicely with the logical model defined for the TDS. This likeness was further highlighted by the fact that the RTserver provides a real-time image to all of its clients showing the current state of the hardware from which it is receiving data. This immediately fulfilled one of the most important TDS requirements.

With the RTie module installed, complex analysis can be performed on data within the TDS, which, when using a distributed system, is at relatively little processing cost. This again proved to fulfil a key project requirement.

Data can be stored for later analysis using RTarchive and can be retrieved using RTplayback. RTplayback uses a standard relational command language (similar to that of SQL) and allows remote command processing, again via any of the common network communications protocols. The technical specification is summarized in Table 1.

TABLE 1: RTworks - Technical Specification

Minimum hardware	*CPU*	All current WS
	Memory	All current WS
	Platform	All current WS
Version	*Development*	YES
	Runtime	YES
Software	*O.S.*	UNIX,VMS
	Programmer's toolkit	YES
	Language	C
Network	*Interface I/O*	User-developed
	IBM TR	NO
	Ethernet	YES
	TCP/IP	YES
	Interface to PC and Macintosh	YES
Support	*Hot line*	YES
Reliability	*System redundancy*	YES
	Diagnostic tool	YES
RTdb	*Maximum no. of tags*	Unlimited
	Update rate	User-written daq
	RAM resident	NO
	SQL compatibility	YES
	No. of task access.	Publish /Subscribe
	Tag reserved by system	NO
	On-line update	YES
	Tag timestamp	YES
	Tag validity flag	YES
	Maximum tag no. retrieval	Unlimited
	Tag query average	User-dependent
	Access protection	YES
Alarming	*Maximum no. of tags*	Unlimited
	Masking	YES
	Alarm severity class	User-defined
	External interface	YES
	Alarm reduction	Inference engine
	Access to reduced alarm	YES
HCI	*Maximum no. of users*	Licence based
	Graphical animation	Dataviews
	X11R5/MOTIF	YES
	Interface to external HCI	YES
	Run time licence	YES
Event detector	*Maximum no.*	User-dependent
	Minimum time interval	User-dependent
	Polling	YES
	Event-driven	YES
Maths & logic	*Interpreted*	Inference engine
	Compiled	NO
	Loop	YES
	Conditional statements	YES
	Block structure	YES
	Data filtering	YES
Trending	*Real-time*	YES
	Maximum no. of curves	Graph-type dependent
	Tag change logging	YES
Data logging	*Interface with ORACLE*	YES
Administration	*Configuration*	YES
	Error reporting	YES

Having thoroughly analysed all the alternatives, we are convinced that RTworks provides the quickest and most cost-effective long-term solution to this project.

Figure 2 shows how RTworks fits into the TCR control environment.

VI. THE TDS AND ITS INTERFACES

A tag is identified by a generic format that describes one point of data monitored by the TDS. The tag semantics is based on an object-oriented approach; it refers to the class of equipment, a specific member within that class, and a specific attribute of the class. When the RTdaq modules receive Dynamic Point Information (DPI) from the equipment interface they convert this information into tags. These tags can belong to one or more datagroups. The RTdaq modules pass these tags to the RTserver which distributes them to the client applications (RThci, RTie, user clients). The inference engine analyses the data it receives and performs actions upon it. The server distributes the tags to all interfaces that subscribe to the datagroup that the tags belong to. Only the real-time image of the equipment is available from the server. Newly added interfaces, such as an application that has just been started, receive the current tags for all the datagroups that they subscribe to. All tag changes sent to the TDS are logged in a 72-hour ring buffer. Tags are time-stamped as closely as possible to the time at which the hardware generates the values. The TDS distributes a standard time in order to synchronize all equipment generating values. Archiving is achieved using the RTarchive module and the RTplayback module is used for data retrieval from the archive.

The TDS interfaces with the following TCR and equipment group systems [5]:

- the equipment control systems
- the Reference Databases (STRefDBs)
- the Central Alarm Server (CAS)
- the HCI applications
- the data and event logging systems
- the TDS Administrator
- the PC/Macintosh environment.

The operations expected of these interfaces will be described in the following sections.

A. Equipment control systems

The Technical Data Server initially interfaces with four types of equipment control systems:

– The MICENE TMS99-95s which have been used since 1984, 150 such units are installed. They control the CERN electrical distribution.

– The BUS Managers are based on a VME rack and powered by a 68030 microprocessor running under OS9. They interface with several industrial GBUSes and 11 such units are currently installed. The BUS Managers are the new generation of MICENE.

– ECATCRs, 34 installed, handle miscellaneous data for the TCR. Each uses a VME bus powered by a 68000 under OS9.

– Industrial systems such as Siemens Programmable Logic Controller (PLC) using the Sinec H1 protocol and Landys & Gyr VISONIK.

In the future there should also be interfaces to the vacuum and cooling systems where a specific protocol is currently used.

The TDS supports both event-driven and polling data-acquisition mechanisms. The acquisition of data from the hardware level is made using specific device drivers which convert the heterogeneous data coming from the various sets of equipment into tags which are handled by the RTdaq. These modules are located on Process Control Assemblies (PCAs). Equipment drivers and RTdaq modules exchange data according to a unique equipment access protocol. This protocol is implemented using TCP/IP sockets. The tag value is stored in a shared database within the RTdaq.

Industrial equipment is capable of sending alarms that can generate automatic commands such as calls to paging systems: These alarms are known as Automatic Triggered Output (ATO). The system that manages the ATOs will be integrated into the TDS architecture.

B. Reference databases

An ORACLE reference database (STRefDB) contains all static and logical descriptions of the points monitored by the TDS in the form of tags. The tags are described in terms of: physical address, description, type, class, member, generated alarm level, etc. Upon initialization and maintenance of the TDS, tag definitions are down-loaded from the STRefDB database to configure the information system.

C. Central Alarm Server

The Central Alarm Server (CAS) is a key software element in TCR operation. It centralizes the alarms generated by the technical infrastructure. Recent investigation has shown a bottleneck in the alarm transmission channel; hence the current transmission system is being reviewed as a part of the TDS project. For the electrical system alarm reduction is currently performed within each PCA, before the alarm is sent to the CAS. This reduction will be moved to the TDS level, since a higher degree of reduction can be performed there. Indeed, TDS will know the current state of all electrical equipment, not only the state of the equipment connected to a single PCA. The TDS will be integrated into the Central Alarm Processing Environment (CAPE).

D. UMMI applications

The UMMI applications consist of the software used by the TCR to survey the equipment. They give a graphical representation of the controlled processes and are animated by data directly acquired from the equipment (polling mechanism). They allow users to send commands to the hardware and they are also used by equipment groups. Some of the applications consist of more than 100 views. The response time is particularly affected by the number of points acquired, the number of PCAs and ECAs involved, and their load. The TDS should improve this situation by offering a rapidly accessible image of the current state of all equipment. User applications will access the TDS with generic C routines that subscribe to the required datagroups. All user commands will be logged. These applications will become event-driven.

E. Data and event logging systems

The Data Logging System (DLS) [6] accesses various types of equipment: electrical, ventilation, water systems, safety, etc. It interfaces with them using a number of logging processes making calls to the SL-EQUIP package [7]. The measurements are performed at a frequency defined in STRefDB with the fastest rate being every 60 seconds. The advent of the TDS will have a large impact on the current logging system.

The connection between the data logging system and the TDS will be very similar to the connection with the UMMIs. Generic C routines will be used to access the tags required. The data logging may also act as a subscriber to tags to be logged.

The Event Logging System (ELS) is still under analysis. Its main purpose is to record events, not declared as alarms, generated by the technical equipment. The system will be able to capture the events at any level of the control system and store them for further analysis. This system may be integrated directly into the TDS architecture.

F. TDS Administrator

This application should offer tools to monitor all internal TDS activity as well as the state of its interfaces. The TDS Administrator will handle, amongst other things, application admissions to the system, tag unavailability, system monitoring, and error identification and recovery.

The TDS Administrator should help to ensure that the TDS will be available 24 hours a day, 365 days a year. Good error reporting and protected access to the TDS will provide improved safety in the working environment. No loss of data should occur in the TDS. This implies the installation of a hot-spare system. The monitoring of the TDS will include the monitoring of its data-acquisition modules which could be widely distributed.

G. PC and Macintosh environment

TDS data will be made available on both PC and Macintosh platforms and could use the promising features of Web browsers. The effective use of the Web will be investigated during the design stage of the TDS. An alternative would be to use a DDE socket. Currently TCR UMMI applications are distributed on PC and Macintosh using Xvision or Mac X.

VII. CONCLUSION

The integration of RTworks into the CERN technical infrastructure will be gradual. The TDS project team should have prototyped the basic functionalities of the overall system by the time this paper is released. The prototype, using RTworks, should include scaled versions of:

- the interface to the STRefDB
- the interface to the equipment
- the alarm reduction
- the CAS interface
- the data logging
- the HCI
- the TDS Administrator.

Approximately 2000 points from three electrical substations will be managed by the prototype. This should indicate which areas of the system will require more work than others and highlight deficiencies in the specifications. Unitechnic-France have committed themselves to the porting of the RTworks data acquisition module to the PowerPC platform running LynxOS.

In accordance with the CERN outsourcing policy, the prototype is being developed in collaboration with STERIA, a company experienced in the field of RTworks integration.

The TDS will change the distribution of responsibility in the monitoring of the CERN technical infrastructure. The TDS will impose a strategy and a common structure for equipment data-acquisition, treatment (alarm, logging), transport and static definition. It will guarantee the distribution of the information to all its client modules. The TDS will provide a common interface to the equipment in order to replace all equipment-specific data servers. CERN equipment drivers and industrial drivers such as the SINEC H1 will exchange data through a standard UNIX communication mechanism with generic RTdaq modules. Nevertheless, the driver that accesses the equipment will stay under the responsibility of the equipment groups.

The TDS adopts a new control system philosophy based on the asynchronous distribution of technical equipment attributes. All TDS components are optimized to provide high performance and reliability.

The task ahead is not an easy one, indeed, it requires a redefinition of a structure that has worked for many years. With new projects such as the Large Hadron Collider on the horizon the technical infrastructure of CERN will have to handle larger quantities of data.

References

[1] "Software Engineering Standards", ESA Board for Software Standardisation and Control, 1994.

[2] D. Sarjantson, S. Lechner and P. Ninin, "Technical Data Server Software Requirement Document", (ST internal document, April 1995).

[3] M. Vanden Eynden, "The New Generation of PowerPC VMEbus Front-end Computers for the CERN SPS and LEP Accelerator Control Systems", CERN/SL 95–74, August 1995.

[4] P. Ninin and P. Sollander, "Utilisation d'un logiciel commercial graphique pour le contrôle des installations techniques du CERN", CERN–ST–MC–TCR/92, October 1992.

[5] D. Sarjantson and P. Ninin,"Technical Data Server, Interface Control Document", (ST internal document, June 1995).

[6] R. Martini, H. Laeger, P. Ninin, E. Pfirsch and P. Sollander, "Data Logging for Technical Services at CERN, these Proceedings

[7] P. Charrue, "Accessing Equipment in the SPS–LEP Controls Infrastructure: the SL-EQUIP Package", SL/Note 93–86 (CO), September, 1993.

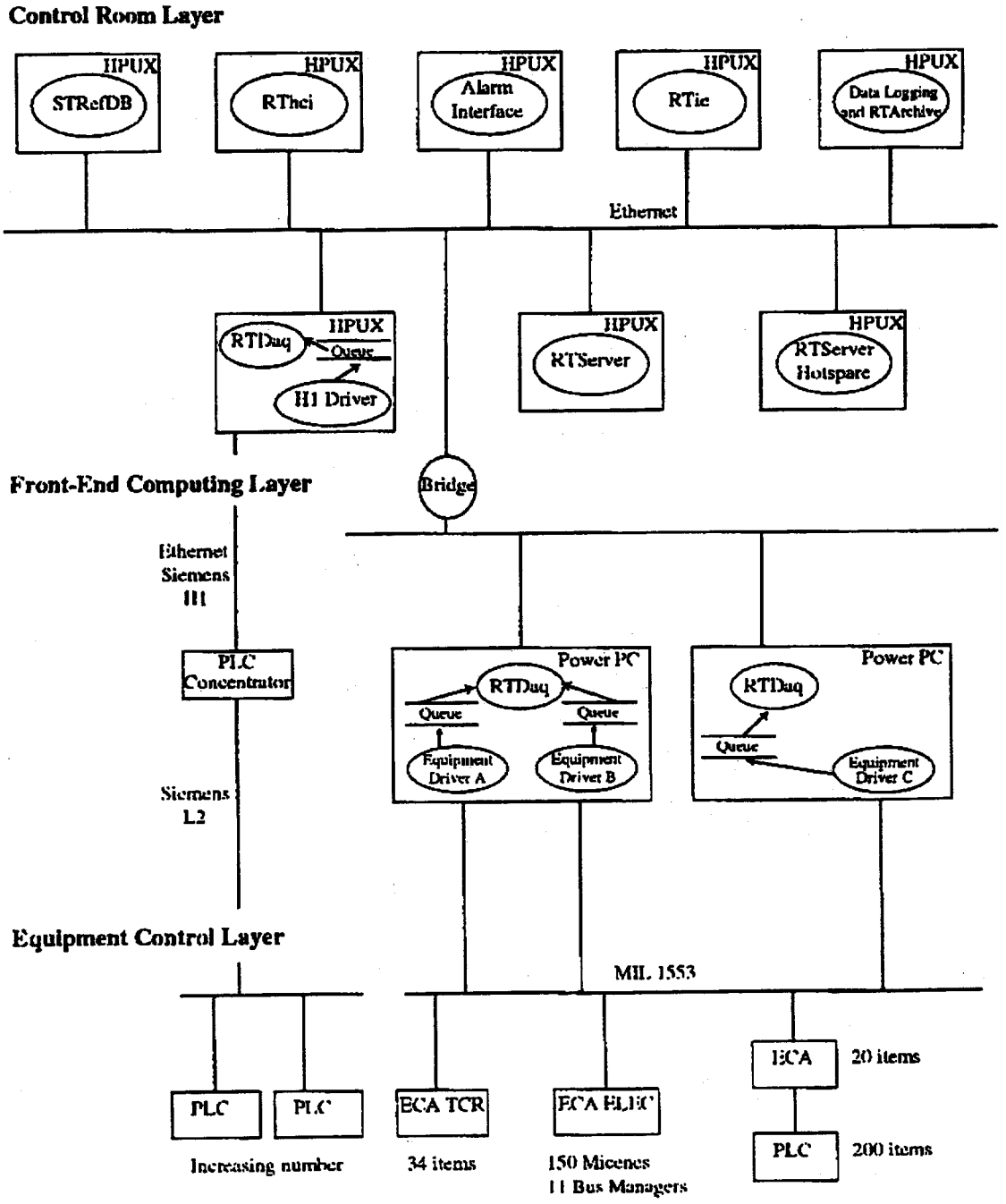


Figure. 2. The Proposed RTworks Structure

Status of ESO Very Large Telescope control software

G. RAFFI
European Southern Observatory (ESO), Karl-Schwarshild-Str. 2
D-85748 Garching bei Muenchen, Germany
Email: graffi@eso.org

Abstract

The Very Large Telescope (VLT) project consists of four 8 m diameter telescopes to be installed at a newESOobservatory in the Atacama desert in Chile. Extensive tests involving hardware and software will start at the Observatory and in Europe next year, but the whole commissioning will extend well after year 2000.

The control software architecture for the VLT project was introduced at the ICALEPCS '91 Conference. The size of the control software has been estimated to be equivalent to a 200 man-years development effort or about 1-1.2 million lines of code. This work is currently being done by ESO staff, partly in collaboration with industry and astronomical institutes.

The ESO team has created a central layer of software on top of a Standard architecture, based on Unix and VxWorks and on a commercial industrial control package, including a real-time database. This layer of software amounts by now to about 500,000 lines of code.

The introduction of an object-oriented event-driven architecture in the control software is one of the recent and most promising developments (reported in a separate paper at this Conference). Other interesting developments are based on the usage of Tcl/Tk in the implementation of a Sequencing tool and of a Panel editor, which allows the creation of homogeneous user interfaces for telescopes and instruments.

The experience and evolution of concepts in the VLT control software are reported, with emphasis on the goal of maintaining consistency while adapting to new technologies.

1 INTRODUCTION

The VLT project consists of four 8 m diameter telescopes, capable of working in parallel as an equivalent telescope of 16 m diameter, to be installed at a new ESO observatory in the Atacama desert in Chile.

The site of the new Observatory is in an advanced stage of construction, while the main mirrors, the telescope structures and all other components, including instrumentation, are being prepared in Europe.

The first components requiring control software will be the dome enclosure of the first telescope, which will be installed in Chile in spring next year. At the same time the first main telescope structure will be tested in Europe, together with its control software.

The first telescope will become operational in two years time, but the entire commissioning period for the four telescopes and the auxiliary telescopes used in the interferometric laboratory will extend well after year 2000. The instrumentation program, involving control software based on the same components and standards, will be developed in parallel and will obviously continue for a much longer time.

2 VLT CONTROL SOFTWARE

The VLT control software consists of all the software which will be used to directly control the VLT Observatory, telescopes and associated instrumentation.

The control software architecture for the VLT project was introduced at the ICALEPCS '91 Conference [1]. It consists of a fully distributed system based on a number of workstations and microprocessors, called LCUs (Local Control Units) (about 40 and 150 respectively in the complete VLT and instrumentation configuration).

This is now in the implementation phase, performed to a large extent directly by ESO staff in the VLT software group, but also by consortia of institutes responsible for some ESO instruments and contractors, who implement some of the telescope subsystems.

Some aspects of the control system architecture, notably the Local area network, the backbone network and the integration of a direct 2Mbit/s link to Europe, will require step-wise upgrades. Digital detectors for optical and infrared observations in particular are very demanding in terms of bandwidth and this is never wide enough. This means that while the global architecture is clear, the media used will have to evolve, and indeed an evolution path from Ethernet to FDDI and ATM is foreseen, although most of the testing and detailed engineering work has still to be done.

3 VLT COMMON SOFTWARE

The main foundation body for the VLT control software is called VLT common software. It consists of a layer of software over the Unix operating system, in the case of workstations, and on top of the VxWorks operating system for the LCU microprocessors. It provides mainly common services, like an architecture-independent message system, a real-time database for all telescope and instrument parameters, error and logging systems and a large number of utilities and tools.

The main Packages in the VLT common software are:

- CCS (Central Control Software) (see also [3]).
  It is a layer of software built on top of a commercial system (RTAP, Real Time Application Platform by Hewlett-Packard), which relys on a real-time database, runnable on several Unix platforms.
- LCC (LCU Common software) (see also [4]).
  It works on LCUs over VxWorks and is completed by a set of drivers for the ESO standard cards and a user interface (GUI) on the host workstation. It is the common platform for all the LCUs of the VLT and instruments. A motor library, dealing with the VLT standard control cards is also contained in this software. Test and debugging tools are also provided.
- HOS (High level Operation Software).
  It consists of a set of high level tools to provide support for operators and astronomers, also used in the preparation phase of Observing runs ahead of time.
- INStrumentation common software.
  Additionally there is a library, which forms what is called the Instrumentation common software and is specific to instrumentation applications.

4 VLT COMMON SOFTWARE RELEASES

It is used across all computers of the VLT observatory (telescopes and instruments) and is designed, implemented and maintained by the VLT software group. The group is also responsible for monitoring software developed outside ESO and for later integration into the VLT control software at the VLT Observatory.

The VLT common software is used in the whole VLT programme, telescope and instruments, by ESO staff, contractors and consortia.

Therefore the VLT software group started a system of releases of this software one and a half years ago, which is by now distributed to about 15 sites, both internal and external to ESO and runs on both HP (HP-UX) and SUN (Solaris 2) platforms. The last release was distributed externally at the end of August 1995.

The VLT common software has a size of about 500.000 lines of code (including code, comments and test procedures), mostly written in C, but the use of C++ is on the increase and Tcl/Tk procedures are also present.

The level of confidence in the capabilities and quality of the VLT common software is quite high at this stage.This does not only result from the application of rules, programming standards, configuration control and test procedures, but mainly from feedback based on field tests. In particular the ESO telescope NTT (new Technology Telescope) is being upgraded to the VLT standards in parallel by an independent but closely linked ESO team.

5 VLT CONTROL SOFTWARE FOR TELESCOPES AND INSTRUMENTS

This is the main part of the VLT control software and emphasis is shifting towards it as more and more design work gets completed. Implementation has started within the VLT software group in several areas, such as telescope subsystems, telescope coordination, CCDs and instrument software prototypes.

The first internal milestone of the Telescope Control System software (TCS) was reached in September, with integration of coordination software and subsystems. This is actually a joint venture between the VLT and the NTT upgrade teams, so that most of the software will be the same. The next major milestone for TCS is coming early next year when TCS has to be ready for tests with the structure in Europe, scheduled for a period of about six months.

Concerning instrumentation and detector software, the detector CCD (Charge Coupled Device) software, after the January 1995 prototype tests, will reach its main milestone in early 1996. This software will be used not only to control and acquire data from scientific CCDs, but also to get images from the "technical" CCDs used for example to drive automatic guiding of telescopes. Real-time display software with a number of interactive features has been developed for this purpose and as part of the VLT common software also.

External teams are well advanced in the development of certain VLT instruments, while ESO staff is also working onothers in parallel. Counting the number of telescope foci available for instruments, one comes to a first generation of instruments close to ten, even without taking possible multiple copies of an instrument into account.

6 VLT CONTROL SOFTWARE TRENDS

A set of new tools in the CCS software (Database Loader, Extended CCS, Event Handler, Class Browser) support an object-oriented design and implementation of the VLT software (workstation part).

The object-oriented Event Handler in particular is one of the recent and most promising developments and is presented in a separate paper at this Conference [2]. It is used as a key element of the Telescope and Instrumentation control software developments, which are on-going at the moment, and enforces an object-oriented design of the surrounding software.

A vendor-independent platform of VLT common software on Unix workstations has also been developed and is called CCS-lite. This does not support access to a real-time workstation database, as the Rtap product is not used in this version of CCS. However it provides access to the LCU database and support at the workstation level for passing messages, errors and logs and allows easy implementation of user interfaces to LCU software

Other interesting developments are based on the use of Tcl/Tk in the implementation of the Sequencer, a tool to support the preparation of an observing program ahead of time, introducing some logic between the observations to be done. This should play an important role in the way to operate the VLT telescopes, where astronomers will not normally be present and will have to describe accurately their observations to a service observer. Tcl, complemented with a number of commands to access message and database services, is then the internal control language of the VLT [6].

Another tool based on Tcl/Tk is the Panel editor (see Figure 1), which allows users to create homogeneous interfaces for telescopes and instruments on the VLT. The basic concept is that it should be easy to develop and change a control panel, but that this should be done according to a given set of conventions. After writing down these conventions, it was then clear that the only way to enforce them would have been to have an editor capable of producing control panels according to them, without requiring programming. The Panel editor, when used within the VLT environment, contains all the hooks to the message system and the database, which are needed.

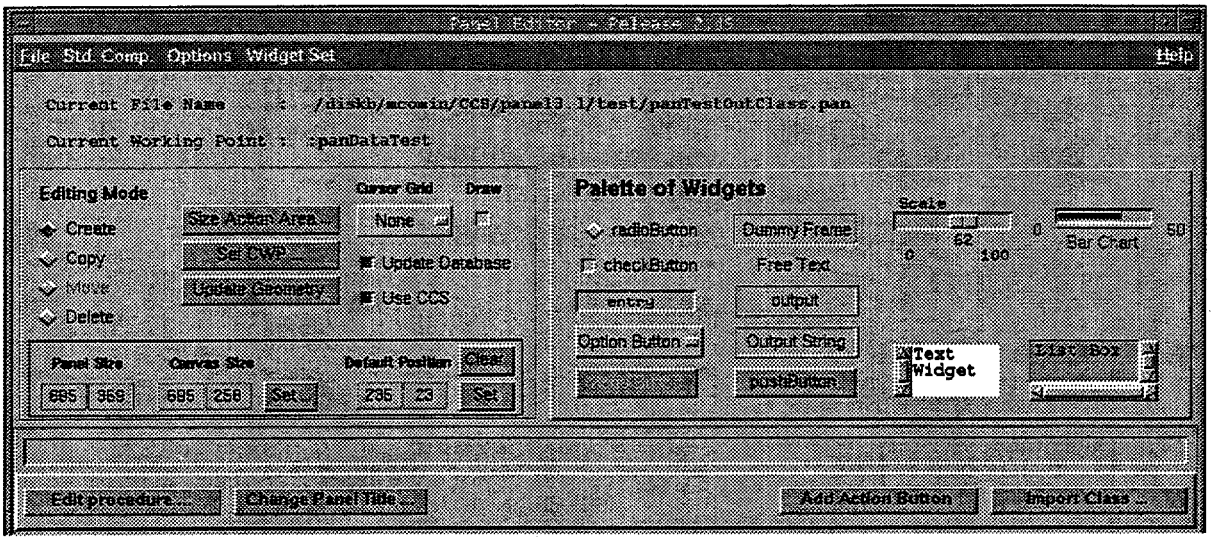


Figure 1 The Panel editor: a tool to develop control panels enforcing a set of conventions.
A stand-alone demo version of the Panel editor can be obtained by the author M. Comin, **mcomin@eso.org**.

7 SOFTWARE ENGINEERING

Software engineering plays an important role in the VLT control software developments.

At first this seemed to be necessarily linked to the introduction of appropriate CASE tools, but after an initial period it was clear that people did not find it an advantage to write functional specifications or design documents according to a given tool. It was a difficult job also for reviewers to get to terms with these tools.

What did remain from this initial phase is rather the correct approach to software development, whereby every software package must follow a given cycle, even if based more on paper and English than on CASE tools, apart from drawings. This means that the functional specifications and design documents must be reviewed and that there must be test procedures and user manuals before a software product is accepted. Measures such as written comment procedures preliminary to reviews, enforcement of standards and lately the introduction of automatic test procedures are all important steps to obtain a coherent and reliable result.

Developers by now realise that all this is right and necessary if they work in a large team, and it has been applied to the VLT common software. We would not have been in a position to distribute that amount of code at the pace of a release every 6 months, with plenty of new functionality each time, without moving more and more towards a more rigorous way to develop software.

The maintenance activity for the common software is taking quite some time, while we have still to develop most of the application software. We use for this a system of SPRs (Software Problem reports or Change requests) and we have already more than 500 this year.

A complete set of specifications and user manuals exists for the VLT software and for the VLT common software in particular. The user manuals by themselves already fill three thick folders.

A first assessment of the early phase of these activities was given at ICALEPCS 93 [5] and it will be interesting to follow the progress in the life of the VLT software.

8 CONCLUSION

The evolution of concepts in the VLT control software has been explained. The experience gained so far has normally been put immediately to use in the next phase of the project. Although a number of developments have already been based on the VLT common software, most of the feedback has still to come, with most of the commissioning activities to be done.

The goal of maintaining consistency in a very distributed development where different groups are involved, has been achieved so far, but integration will show how successful this was. This has not been based on a set of unchange-

able rules and standards, but on a pragmatic approach of following innovative techniques when affordable, but avoiding on purpose to be really the first ones to try them out. This way of keeping in touch with evolution, while being conscious of the priorities and deadlines of our project, is also necessary when long commissioning times spanning several years are involved.

For people who want to know more about the VLT project, access to ESO VLT software documentation can be gained via the Web (under **http://www.eso.org**).

9 ACKNOWLEDGMENTS

It is a pleasure for me to acknowledge the contributions of my colleagues in the VLT software group who have been the authors of most of the software that I have mentioned in this paper.

10 REFERENCES

[1] G.Raffi - Control Software for the ESO VLT - Proc. of ICALEPCS, Tsukuba, 1991, KEK Proc. 92-15

[2] G. Chiozzi - An object-oriented event-driven architecture for the VLT Telescope Control Software - These Proceedings

[3] B.Gilli - Workstation environment for the VLT - Proc. of SPIE, vol.2199, pp.1026-1033, 1994

[4] B.Gustafsson - VLT Local Control Unit Real Time Environment - Proc. of SPIE, vol.2199, pp.1014-1025, 1994

[5] G.Filippi - Software engineering for ESO's VLT project - Proc. of ICALEPCS, pp.386-389, Berlin, 1993

[6] E.Allaert, G.Raffi - The VLT control software - Local and remote support for operations by astronomers and operators - ESO Technical preprint No. 69, 1995

The Gemini Control System

Richard J. McGonegal and Stephen B. Wampler

The Gemini 8-m Telescopes Project

950 N. Cherry Ave., Tucson AZ 85726

ABSTRACT

The Gemini 8-m Telescopes Project has been charged with extremely challenging performance requirements in the areas of tracking, pointing, image quality and operations. In addition the work is being done in a distributed fashion on 3 continents - which itself puts constraints on the design. The author will describe the software, hardware and control components of the Gemini system as well as reporting the predicted performance of the design.

Keywords: telescopes, performance, control systems

INTRODUCTION

The Gemini Project is an international partnership to build two 8-meter telescopes, one on Mauna Kea, Hawaii, and one on Cerro Pachon, Chile . The telescopes and auxiliary instrumentation will be international facilities open to the scientific communities of the member countries. The international partnership is made up of the United States, the United Kingdom, Canada, Chile, Argentina, and Brazil. The telescopes will be high performance, 8-meter aperture optical/infrared telescopes and have a planned completion date of 1998-2000. The goal of the telescopes is to exploit the best natural observing conditions and to undertake a broad range of astronomical research programs within the national communities of the partner countries. In order to reach this goal Gemini has set demanding requirements in terms of image quality, tracking, pointing, and availability (Table 1)

SCIENCE REQUIREMENTS

Performance Requirements

Table 1 shows the performance required of the Gemini telescopes during operation.

TABLE 1. Gemini Performance Requirements

Specification	Requirement	Description
Image Quality	0.1 arcsec	increase in 50% encircled energy diameter
Tracking	0.044 arcsec	RMS jitter in line of sight
Pointing	3.0 arcsec	in service pointing
Availability	98%	time collecting science photons

Image quality is defined as the variation in the image point spread function integrated over one hour. Image quality is determined by measuring the 50% encircled energy diameter (eed) of a stellar source. Image quality can be quantified by comparing long and short exposure images. Tracking is defined as the jitter in the telescope line of sight over one hour Tracking is determined by measuring the instantaneous (or nearly so) centroid of a stellar source. Tracking can be quantified by taking repeated centroid measurements and calculating their root mean square. Pointing is defined as the ability to align the telescope line of sight to a particular position on the sky. Pointing is determined by measuring the difference between the observed and predicted positions of a stellar source. Pointing can be quantified by taking repeated measurements across the visible sky and taking their root mean square. Availability is defined as the amount of time spent doing science. Availability is determined by measuring the amount of time spent collecting photons, including required calibrations. Availability will be quantified by tracking the long term average of the measured time compared to the total time

available. The requirement for availability will not be addressed in this paper. It is the subject of ongoing work at the project and will be reported on at a later date.

In all of the above the requirement is a residual reached after a number of systems are correcting the telescope. The system makes extensive use of :

- look-up-tables (LUTS) which position the telescope axes and optics in real time
- active feed back of a focal plane guide star's tip/tilt/focus measurements - which drive the secondary
- active feedback of a focal plane star's higher order zernike measurements - which correct the figure of the primary mirror

Operational Requirements

The Gemini telescopes must also support a wide range of operational modes:

Classical Observing - user manually sequences the different telescope subsystems to acquire data
Preplanned Observing - user plans detailed use of facility in advance and submits this to facility
Queue Scheduling - parts of one observer's program are interspersed with those of others in order to make more efficient use of facility
Flexible Scheduling - programs to be executed are selected depending on environmental conditions in order to make efficient use of existing conditions
Service Observing - program is executed by a member of observatory staff - not the Principal Investigator

These different modes create a requirement for a programmable upper layer to the software - which Gemini calls the Observatory Control System. This system acts to synchronize and sequence the actions of the different subsystems which make up the observatory system.

SOFTWARE

User View

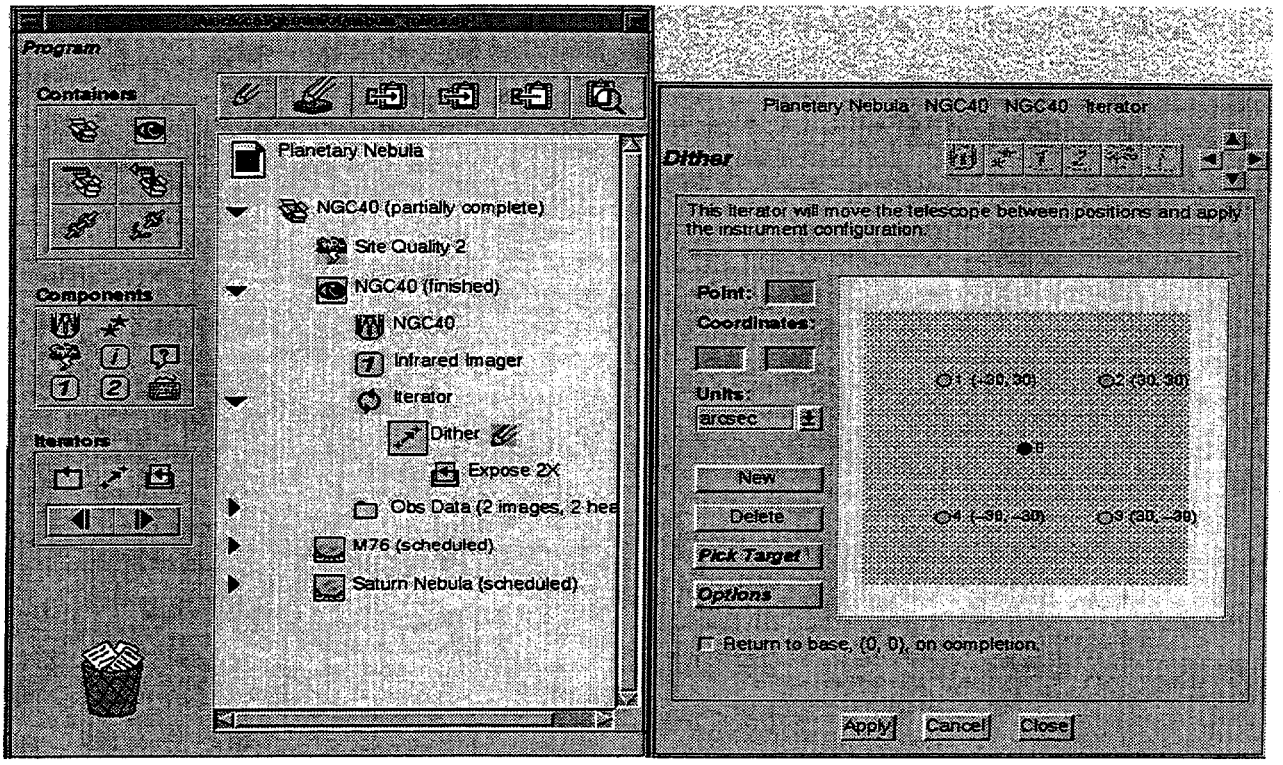
Operation of the Gemini telescopes is through the use of *Science Programs* developed by the astronomers. These programs provide a means for astronomers to describe how they want the system to perform in a structured, hierarchical outline that is executable by the Observatory Control System. One goal of the design of the Science Programs is to provide the flexibility required for interactive use of the telescopes with the structure required for efficient planning and scheduling. The Science Program is used by the Observatory Control System to identify resource demands (which system features are needed and how long they are needed), environmental constraints (clarity of 'seeing' required, wind conditions), timing constraints (when is an observation possible), and an ordering of actions (which observations to take first, and which concurrent operations should be permitted). The system can then use this information to schedule the program with other programs to keep the telescope fully utilized. Observatory staff can create, monitor, and adjust nightly *plans* consisting of a number of such programs.

While the Science Program provides structure, embedded within a Science Program are *consoles* where astronomers have the flexibility to configure the telescope equipment for their specific needs. Figure 1 shows a prototype science program with an open console where the astronomer is setting up a pattern of target positions where images are to be taken (a *dither* pattern). In practice, a star field would typically appear in the positioning window.

An important feature of the software interface is that the same consoles used when developing programs are the same ones available during operation of the telescope for making adjustments during an observing session. For example, the astronomer can reopen the above console to make fine-tuning adjustments to the target positions after the telescope is pointed in the proper direction.

Since the observing process is often highly interactive, with the astronomer potentially making a large number of such *fine-tuning* adjustments, a Science Program is a dynamic object that is itself transformed during the observing process it represents. In the example shown above, parts of the program have completed and the program now includes data that has been collected (found in the *Obs Data* folder in the program). In addition, the program shows the status of the observing process it describes.

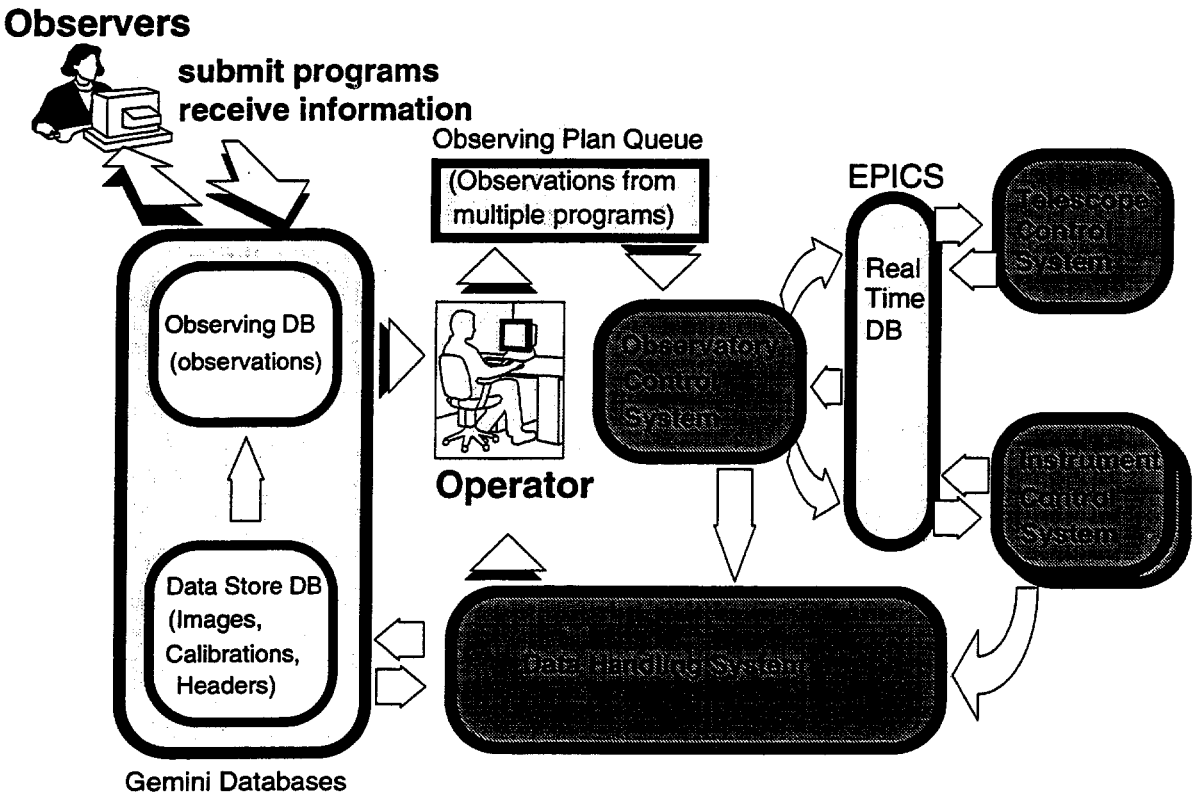
FIGURE 1. Science Program with Console



Software Organization

The Gemini Control System is implemented as four major systems, as shown in Figure 2.

FIGURE 2. The Gemini Principal Systems



Each major system is responsible for some specific aspects of control during operation and may be split into one or more subsystems to accomplish this task. The *Observatory Control System* manages all aspects of the science program and sequences the interactions between the other major systems while a science program is being performed. In addition, the Observatory Control System supports planning activities by providing tools with which the observatory staff can quickly match science programs to resources and existing environmental conditions. The *Telescope Control System* is responsible for the pointing and tracking the telescope and all related subsystems, such as the carousel (rotating portion of the enclosure building), the mount, the primary and secondary mirrors, and the cassegrain rotator (rotates to keep image orientation constant as telescope tracks across the sky). It is the Telescope Control System that has primary responsibility for maintaining the best possible image quality on the Gemini telescopes. The *Instrument Control Systems*, one per instrument (Gemini typically has three science instruments mounted at any one time). These are responsible for each instrument's mechanical/optical components and the detector control. The Instrument Control System has primary responsibility in the acquisition of useful data from the images provided by the telescope. Having multiple active Instrument Control Systems allows the Observatory Control System to provide parallel sequencing. For example, one instrument might be collecting internal calibration data while another instrument has the telescope beam. The *Data Handling System* collects data from the instruments, associates relevant status items with the data, and provides both quality-control feedback to the astronomer and preliminary analysis to remove instrumentation and environmental effects from the data. The Data Handling System is also responsible for archiving the data and associated status items.

The Telescope and Instrument Control Systems are based on EPICS. The Observatory Control System is responsible for mapping the information in a Science Program into the database-driven domain of EPICS.

Principle of Operation

The Gemini Control System views the overall system as existing in a particular *state* at any particular moment. The Observatory Control System transitions the system from one state to another by supplying *configurations* that describe the controllable conditions for the new state. So, systems are controlled by being directed to achieve specific *target* configurations. The Observatory Control System can monitor the performance of each system by comparing the target configuration with the *actual* configuration as reported by the other systems. This approach maps well onto EPICS-based systems and can be adapted to non-EPICS systems as well. The science consoles found in Science Programs provide a direct means of specifying configurations, where each console contributes a set of *attributes* and their associated *values* to a configuration. These attribute/value pairs can be directly mapped onto the process variables found in an EPICS database.

Because the major system components are separate systems, resources can be allocated efficiently. For example, an instrument that has been given access to the telescope beam may relinquish that access as soon as photon collection is complete. The Observatory Control System can then begin concurrent execution of another Science Program (or another part of the same science program) and begin moving the telescope to a new target position as the instrument reads out its detector and transfers data to the Data Handling System. (During Classical Observing it is possible that the astronomer will have disallowed this advance motion, performing a quality check of the data to determine if more data needs to be collected at the current target position.)

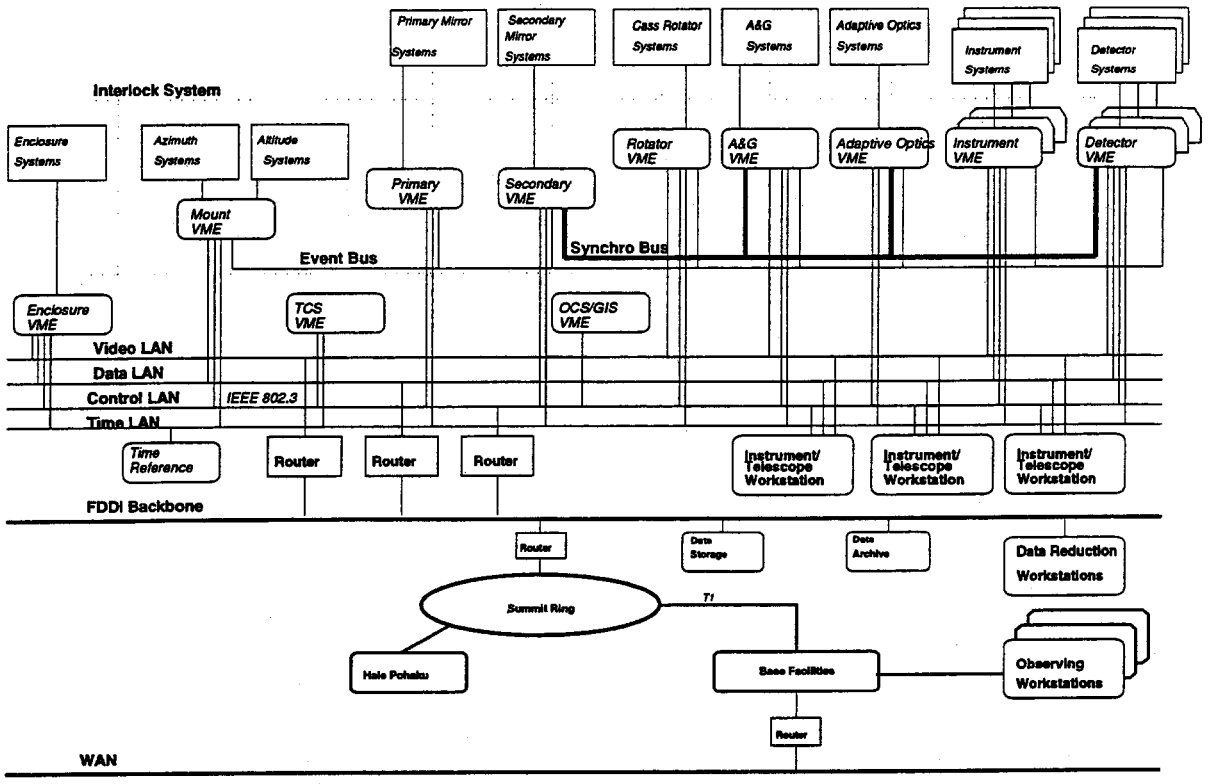
HARDWARE

The Gemini Control System is a distributed system using both Unix workstations and VME crates running EPICS and VxWorks. These components are connected with several LANs. Figure 3 shows the hardware layout for the Mauna Kea telescope. (The Cerro Pachon telescope is configured similarly.)

The VME crates are all EPICS based, although some instruments may not be using either EPICS or VME systems for control. To prevent the high volume of data from interfering with time critical control commands, system command and data flow are provided with separate LANs. A third, very high speed connection, called the *Synchro Bus* is provided for the extremely time-critical control signals from wavefront sensing probes to the secondary mirror. The *Event Bus* is a collection of analog signal connections for synchronizing instrument detectors to quick telescope movements. Finally, an Interlock System both detects and prevents dangerous situations.

Two VME crates are not connected to any hardware devices - the TCS VME crate is used by the Telescope Control System to implement the Gemini pointing and tracking models and to synchronize the TCS subsystems. The OCS VME crate serves as a repository of system *status* information from other systems. This EPICS system provides a common collection point and to avoid monitoring actions from impacting actual control. It also provides a way for non-EPICS systems to provide status information back through EPICS.

FIGURE 3. Mauna Kea Hardware Layout



CONTROL

Control Systems Available

The following control systems are available for use in the different modes of operation:

- mount
  - —altitude drives (4)
  - —azimuth drives (8)
  - —cassegrain rotator (4)
- secondary
  - —axis articulation
  - —fast tip/tilt/piston
- primary
  - —passive pneumatic air bag providing 80% of axial support with controlled pressure
  - —passive hydraulic wiffletree providing 20% of axial support with controlled tip/tilt
  - —passive hydraulic lateral support with controlled translation
  - —active axial and lateral pneumatic support
- adaptive optics
  - —deformable mirror
  - —tip/tilt mirror

Sensors Available

The following sensors are available for use:

- *fast wave front sensor* - provides low spatial order, tip/tilt and focus, error at up to 200/15 Hz respectively in parallel with science observing

- *slow wave front sensor* - medium spatial order (~ 30 spots) read out once per minute to provide tip/tilt, focus, astigmatism, and coma in parallel with science observing
- *calibration wave front sensor* - provides high spatial order (~400 spots) map of wavefront errors; precludes science observing while in beam
- *adaptive optics wave front sensor* - provides medium spatial order map of wavefront errors at very high rates (< 1 KHz)
- *tape and friction encoders, fiducial system*
- *laser measurement of secondary position, tilt meters, accelerometers, strain gauges* (all TBD)

Servo Bandwidths

The servos have bandwidths as follows:

TABLE 2. Servo Bandwidths:

Servo	Sensor	Max Servo Bandwidth (Hz)	Max Sampling Rate (Hz)
Closed Loop	Fast WFS	0.100	1-10
Tip/Tilt Loop	Fast WFS	40.000	200.000
Fast Focus Loop	Fast WFS	3.000	15.000
Active Loop	Slow WFS	0.003	1/60
Adaptive Loop	Adaptive WFS	150.000	750-1000

Operating Modes of the Telescope

The normal operating mode of the telescope will be with the tip/tilt loop, fast focus loop, and active optics loop closed - this is refered to as Active Optics Mode.

Open Loop Mode

In open loop mode there is no star on the wavefront sensor and the primary active and passive system run from LUTs. There will be LUTs available which will be based on a certain grid spacing on the sky and the respective servos will interpolate within these grids. This interpolated value will be used to command the next position set of the active actuators and the passive system. In general, the LUTs will be based on 10 degree grids in right ascension and declination and the servo loops will determine new positions at a 20 herz rate. If required there will be temperature corrections for the LUTs. These LUTs will be used to remove the repeatable deformations of the primary based on the current position and temperature. We assume that the LUT acts as a high pass filter with a break at 0.003 Hz (5 minute time scale).

The dominant contributions to image degradation during open loop mode are gravitational warping of the mirror cell and higher order (focus and above) atmospheric effects.

Closed Loop Mode

In closed loop mode a guide star is available for making corrections but there is no fast tip/tilt of the secondary available. The position of the guide star in the focal plane is used to make relatively slow (< 0.1 Hz) corrections to the position of the mount, primary, and secondary which are required due to (a) errors in the LUTs and (b) non-repeatable errors such as hysterisis and wind shake. For the purpose of this discussion we will define slow as 0.1 Hz. We will not try and make corrections to the mount or the secondary system faster than 0.1 Hz. Limit on mount correction bandwidth will not be sampling of guide star position but rather the maximum speed at which the mount can be moved without inducing vibrations. During this mode all corrections will be performed by slow tip/tilting of the secondary. If required, the DC offset of the secondary slow tip/tilt mechanism will be used to make corrections to the mount.

During closed loop mode the LUTs will be used to feedforward position and velocity information to the servo system. The guide star information will be used to make corrections to the LUT positions. We model this as a closed loop servo with a bandwidth of 0.1 Hz. This servo improves the windshake performance but the major contributors to image centroid motion are still windshake and atmospheric tip/tilt.

Tip/Tilt Secondary Mode

In tip/tilt mode a guide star within the isokinetic patch is used to generate corrections (at up to 200 Hz) to the tip/tilt of the secondary in order to maintain the image centroid. The main sources of error to be corrected are wind shake of the telescope/optics and atmospheric turbulence induced motion of the image centroid. This information can come from a dedicated X/Y guider or it may come from the wavefront or curvature sensor. In this mode of operation the adaptive optics system is not operating.

During this mode all corrections will be made by tip/tilting the secondary. If required the DC tilt offset of the secondary will be used to inject corrections to the slow secondary 5 DOF mount and the telescope drives via low pass filters. The LUTs will be used as in closed loop mode. It is the goal of the LUTs to model the repeatable errors such that DC corrections are unnecessary. It is the goal of the telescope design that it is stiff enough that dynamic errors and non-repeatable errors are small enough such that corrections are unnecessary. We model this as a closed loop servo with a bandwidth of 40 Hz and a sampling rate of 200 Hz. The bandwidth is driven by the requirement to reduce the windshake to fit within the error budget. The sampling rate is driven by the requirement to have a sampling rate that is at least 5x the servo bandwidth in order to have a stable servo system. Once the tip/tilt system is activated the dominant contribution to encircled energy diameter is the higher order effects of the atmosphere.

Fast Focus Mode

In this mode the star which is being used to generate tip/tilt information is also used to provide focus information at a reduced rate (estimated to be 15 Hz). This focus information is used to make corrections to the focus position of the secondary by making offsets in the actuators of the secondary tip/tilt system. We model this as a closed loop servo with a bandwidth of 3 Hz and a sampling rate of 15 Hz. The effects of a changing focus do not have large effects in open loop or closed loop mode because these modes are dominated by wind shake. The improvement in image quality due to the tip/tilt system results in the focus effects being a significant contributor to image quality. The major impact on telescope focus is thermal changes in the optical support structure - these can be adequately reduced by thermal sensors and calibration - however extra margin is introduced with the fast focus system. Wind buffeting and atmospheric higher order effects (beyond tip/tilt) cause a significant contribution to image quality relative to the error budget. These effects can be reduced substantially by the use of a fast focus system with a 3 Hz closed loop bandwidth.

Active Optics Mode

This is the normal operating mode of the telescope and it includes look up tables, fast tip/tilt and focus described in previous operating modes. The coma component is used to correct translation of secondary mirror and astigmatism and higher components used to correct figure of primary mirror via LUTs. In this mode an off-axis star (a star outside of the isokinetic patch must be integrated long enough (~60 sec) to remove effects of atmospheric seeing) is observed with a wavefront sensor and the information derived from that signal is used to make corrections to the primary figure. The adaptive optics system is not used in the active optics mode. During this mode all higher order corrections will be made by altering the figure of the primary. If required the DC offset of the primary figure may be used to make corrections to the 5 DOF mounts of the primary and secondary - but it is the goal of the appropriate LUTs to model the repeatable errors such that corrections are unnecessary. It is the goal of the primary support system that it is stiff enough that dynamic errors and non-repeatable errors are small enough such that corrections are unnecessary. The primary axial LUT will be used during this mode to feedforward corrections to the primary figure. The wavefront sensor information will be used to make corrections to the LUT and to update its current value. The LUT will be capable of working in both an absolute mode and in an incremental mode. All tilt and wavefront information from the Acquisition and Guide unit (A&G) will be rotated to apply to the fixed reference frame of the primary mirror. This will be handled by the Telescope Control System as it collects all of the relevant position and orientation data.

Adaptive Optics Mode

In this mode a bright star is used as a probe of the wavefront errors at a high sampling rate. This information is used to manipulate an internal tip/tilt mirror and a deformable mirror provided with the AO system. The AO deformable mirror with tip/tilt optic used to extend tip/tilt corrections to higher frequencies and wavefront correction to more modes. During this mode all tip/tilt corrections will be made with the AO small internal tip/tilt mirror. If required the DC offset of this mirror will be used to make corrections to the tip/tilt position of the secondary. This in turn will make corrections to the mount if needed.

- DC position of tip/tilt optic used to correct secondary tip/tilt position via a low pass filter
- DC focus term of deformable mirror used to correct secondary focus via a low pass filter
- DC figure terms of deformable mirror used to correct primary figure via a low pass filter

During this mode all wavefront corrections will be made with the deformable mirror. If required the DC offsets of the deformable mirror can be used to make corrections to the primary figure, primary 5 DOF system, and the secondary 5 DOF system. The appropriate LUTs will be used in this mode to feed forward position and velocity. However the secondary tip/tilt system will not be used other than as a follower to the adaptive tip/tilt mirror DC position. The primary axial support LUT will be used in this mode to feedforward corrections to the primary figure. The curvature sensor information will be used to make corrections to the LUT position and to update its current value. The LUT will be capable of working in both absolute and incremental modes.

PERFORMANCE ANALYSES

Initial Conditions

In order to perform the analyses it is necessary to pick a given set of conditions to use for the performance estimate. The standard case used throughout is:

- observing at 2.2 microns and guiding at 0.7 microns
- the existing atmospheric conditions are the upper 10^{th} percentile - equivalent to an R_0 of 40 cm at 0.5 microns
- there is a single dominant seeing layer 4 km above the site with a mean wind speed of 20 m/s in that layer
- the tip/tilt information is coming from a zero read noise detector within a 1.5 arcminute radius of the science object - a star brightness is chosen such that there is a 90% chance of finding one in this field
- the higher order Zernike corrections are coming from a 5 e^- read noise device within a 7 arcminute radius - a 995 sky coverage is assumed for this larger field
- the external wind speed at the enclosure is 11 m/sec
- the telescope is at 45 degrees zenith angle and perpendicular to the wind direction

In addition performance has been estimated in a) low wind speeds, 3 m/sec, and b) median seeing, R_0 of 22 cm. at 0.5 microns.

Image Quality

The current estimate for the image quality of the telescope is 0.102 acseconds increase in the 50% eed. This is well within the error budget at 45 degrees zenith angle, 0.123 arcseconds, and is very close to the zenith requirement of 0.100 arcseconds.

TABLE 3. Top Level Image Quality Error Estimate

Area	Raw	Open	Autoguider	Tip/Tilt	Focus	Active
Optical Design	0.065	0.065	0.065	0.065	0.065	0.065
Surface Errors	0.201	0.187	0.187	0.140	0.105	0.046
Optical Alignment	0.014	0.014	0.014	0.014	0.014	0.014
Self Induced Seeing	0.027	0.027	0.027	0.027	0.027	0.027
Dynamic Alignment	0.147	0.036	0.036	0.025	0.004	0.004
Tracking	3.676	0.732	0.332	0.056	0.056	0.056
RSS Totals	3.685	0.760	0.389	0.169	0.139	0.102

Tracking

The current estimate of tracking performance is a 0.037 arcsec jitter in the telescope LOS which translates to a 0.056 arcsec increase in the 50% eed. This is slightly above the current error budget of 0.044 arcsec 50% eed.

TABLE 4. Top Level Tracking Perfromance Esitimates

Area	LOS jitter (arcsec)	50% eed (arcsec)	Error Budget
Wind Shake	0.024	0.036	0.041
Measurement Error	0.017	0.025	0.006
Off Axis Guiding	0.009	0.013	0.006
Non Linear Effects	0.022	0.033	0.014
RSS Total	0.037	0.056	0.044

Pointing

The overall pointing performance of the telescope is estimated to be 2.43 arcseconds RMS. The breakdown of this relative to the error budget is seen in the table below.

TABLE 5. Top Level Pointing Error Estimate Cause

	Before Correction	After Correction	Error Budget
Calibration Stars	0.10	0.10	0.10
Interpolation Function	0.50	0.50	0.50
Encoder Repeatability	0.14	0.03	0.15
Position of Optics	58.73	2.02	2.67
Position of Mount	16.06	1.25	1.25
RSS Total	60.89	2.43	3.00

CONCLUSIONS

- The current Gemini design meets the demanding requirements set by the scientific mission of the telescope.
- In order to calculate performance estimates in this regime it is necessary to use the best modeling tools and expertise available.
- A range of initial conditions must be explored to fully understand the implications of a given design on performance.

ACKNOWLEDGMENTS

This report would not have been possible without the concerted efforts of the entire Gemini engineering team. I would like to give special thanks to M.Burns, M.Sheehan, B.Ellerbroek and J.Oschmann for all their model runs and comments on preliminary versions of this report.

REFERENCES

[1] D.Rockey, Absolute Pointing Error Budget, Keck Observatory Report No. 159

[2] M.Burns, Servo Simulation and Modeling for the Gemini 8-m Telescopes, SPIE Proceedings V.2479, 1995

[3] R.Racine, private communication, 1994

The DØ Experiment Significant Event System

S. Fuess, S. Ahn, J. F. Bartlett, S. Krzywdzinski, L. Paterno
Fermi National Accelerator Laboratory

L. Rasmussen
State University of New York, Stony Brook

Abstract

A Significant Event System for the DØ Experiment Online Data Acquisition (DAQ) system has been operational since 1992. The system collects and distributes messages related to alarms, heartbeats, and DAQ state transitions. In this paper we give an overview of the hardware and software elements of the system, describe the data flow, give details of the message structure and individual applications, and present an example which illustrates the operation of the system.

I. HARDWARE ELEMENTS

The Online Data Acquisition (DAQ) System for the DØ Experiment was developed with two independent data paths: a high speed customized uni-directional event data path and a standard network bi-directional control and monitoring path [1,2]. A major software component utilizing the monitoring path is the *Significant Event System*, which manages alarm, heartbeat, and run-state transition messages.

There are three principal hardware components of the monitoring path which contribute to the *Significant Event System*: Front End systems, a connecting network, and a Host cluster. The Front End systems, which act as the interface to the detector and other environmental monitoring and control devices, are based upon the Fermilab LINAC control system [3,4,5]. The majority of approximately 35 Front End systems are based upon a 68020 processor residing on a Motorola VME133A card, accompanied by a memory card, a Token Ring interface card, and a utility card which drives an external monitor. Each processor has access to the VME bus of the crate within which it is located, access to other VME crates via a Vertical Interconnect bus extender, or access to other monitoring devices via a MIL-1553B serial link. The remainder of the Front Ends are individual IBM PC systems, which acquire information via various external connections. All of the Front Ends operate by continually repeating (at 15 Hz for the VME based systems) a cycle of data acquisition to fill a local data pool, compare the readings to a local database of analog nominal and tolerance or binary nominal values, and generate and/or process messages. Each Front End system can monitor several thousand analog and binary channels.

The Front End systems are all connected to a Token Ring network. A set of three identical and parallel Gateway nodes act to connect the Token Ring to the Ethernet network used by the remainder of the control and monitoring path elements. Each gateway is a single-board MicroVAX computer running the VAXELN operating system. The peak capacity of each node is approximately 50 to 80 kilobytes per second, depending upon the record size. At peak load, during the trigger condition downloading phase of running, the Token Ring LAN operates at approximately 30% of capacity.

The Host system for the DØ Experiment is a VAX and Alpha mini-computer and workstation cluster running the VMS operating system. In addition to the event data acquisition tools, a suite of applications dedicated to the collection and monitoring of *significant events* runs on this system. These applications will be described in Section 3.

II. DATA FLOW

Figure 1 illustrates the data flow for the *Significant Event System*. The central application in the system is the *Alarm Server*, to which all messages are sent and from which all messages are distributed. There are several message types passed among cooperating applications, the most important of which is the *significant event* message (also referred to as *alarm* message). The other message types are heartbeats, filter profiles, database requests, database replies, and run-state control information. The latter group of message types are specific to certain tasks, whereas the *significant event* message is a more general form which serves multiple purposes.

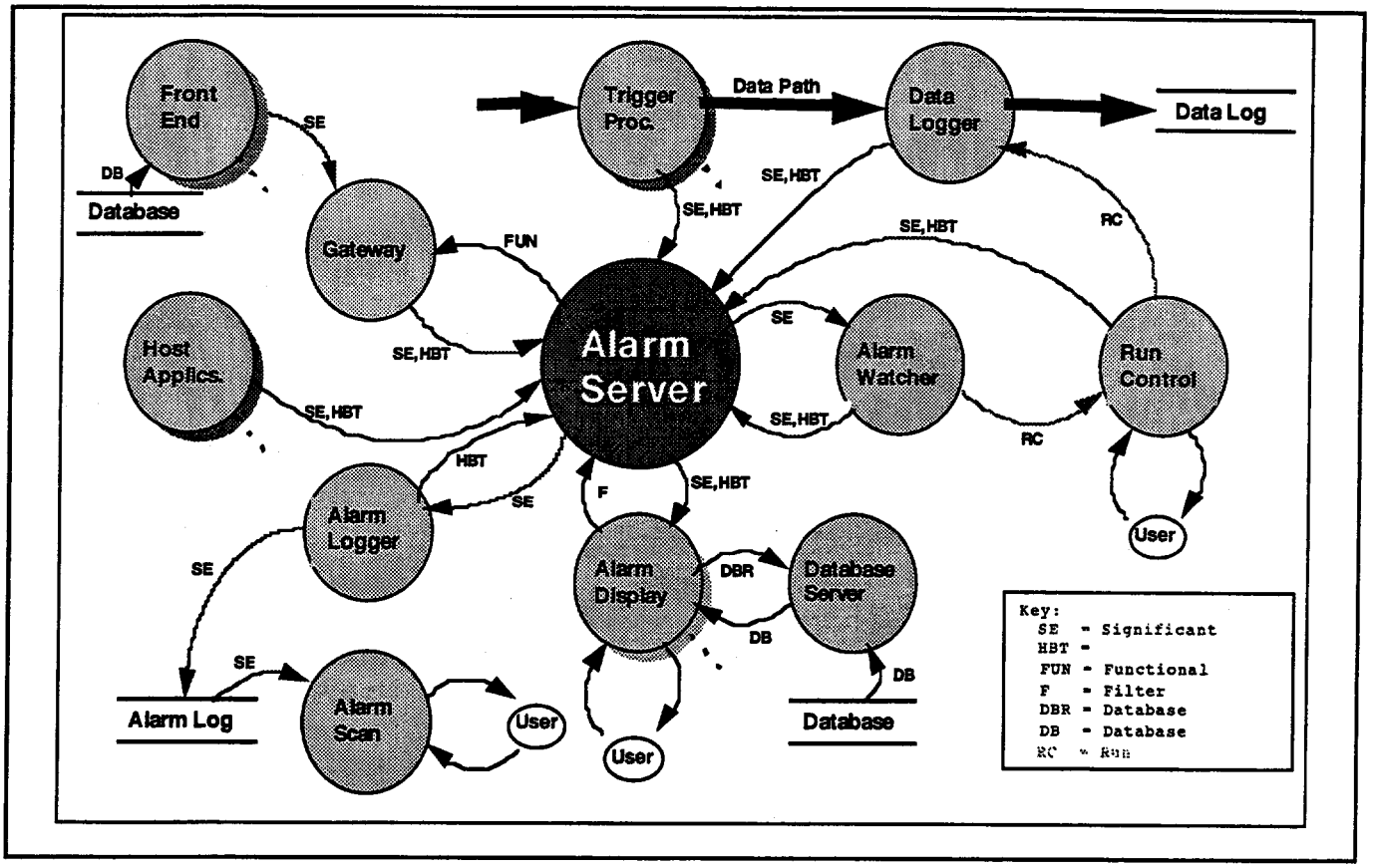


Figure 1: Data Flow of Significant Event System

A *significant event* message contains the following information: a state transition code indicating a good to bad transition, bad to good transition, or informational only; a device and an attribute name; the front end system (or other application) identity and local identifier; a corresponding database identifier; a priority value from 0 (low) to 255 (high); the time and date; and a supplementary block which depends on the nature of the device. For *analog* devices the supplementary block contains the nominal, tolerance, and current readback values. For *binary* devices, the supplementary block contains the nominal bit value and the current readback. There is also a *comment* class of devices (principally used by software applications) which has a 128-character string in the supplementary block.

Figure 1 indicates the flow of *significant event* and other messages. The next section describes the applications that produce and consume these messages.

III. APPLICATIONS

There are three classes of applications within the *Significant Event System*. They are the *significant event* generators, the Gateway, and the *significant event* consumers.

A. Significant Event Generators

Significant events are generated by Front End systems, Host applications and other software applications within the Data Acquisition path. The Front End processors, upon noting readings which are inconsistent with their local databases of nominal and tolerance values, place messages on the Token Ring. Such messages are multicast with an identifying group functional code. On the Token Ring the *significant event* messages are framed within the ACNET (Fermilab Accelerator network transport) protocol, which allows for an accompanying data format block indicating the elemental composition (bytes, words, long words, quad words, floating point, and strings) of the message.

A major component of the data path is a set of processors running the software trigger code. These processors are VAX workstations using the VAXELN operating system, and executing FORTRAN and PASCAL reconstruction and filter code. A library of routines callable from these languages within ELN is provided by which *significant event* messages can be generated and transmitted by DECNET to the Host system.

Applications can also generate *significant event* messagesn the VMS Host system. A set of routines for VMS is provided to generate and transmit the messages by mailbox (local) or DECNET (remote). The typical suite of applications includes run control, event logging, event monitoring and detector monitoring tasks.

B. Gateway

The Gateway processors also run the VAXELN operating system. The purpose of these nodes is to provide the interface between the Token Ring and Ethernet physical layers and additionally between the ACNET and DECNET protocols. As previously indicated, ACNET protocol messages have a format block that describes the internal data structure. The Gateway tasks use this information to perform the appropriate conversions to account for the byte order and floating-point representation differences between the Front Ends and the VAXes.

The Gateways maintain independent logical connections to all DECNET clients, including the important connection to the *Alarm Server* task. Each client is allowed to select the addressing modes of Token Ring messages in which it is interested; the *Alarm Server* picks the messages with the group functional code assigned to significant event messages, and hence sees only such messages. The Gateway tasks also buffer incoming and outgoing messages for each remote client, and hence improve the overall bandwidth on each logical circuit.

C. Significant Event Consumers

A set of applications exists on the VMS Host system to process *significant event* messages and to provide information to the detector users. These applications are written principally in PASCAL with some FORTRAN. All are based on a common layered structure, with application specific routines calling routines from a generalized client / server package, which uses an InterTask Communication package, built upon either asynchronous VMS mailbox (local) or DECNET (remote) task-to-task communication.

The client / server package provides a common framework in which the internal message buffering and queuing, error handling, and monitoring actions are provided for the shell application. The basic element of the package is the logical circuit, with utility routines to establish and break network connections and transmit messages. Customized callback routines may be specified to handle any abnormal condition. In the current implementation all activities are queued asynchronously and processed synchronously. In a future implementation each circuit's activities will occur within an independent POSIX thread of the application.

The client / server package includes several features which contribute to the robustness of these applications. The first feature is that of guaranteed message delivery from the server to clients. Any message that cannot be immediately and successfully transmitted is retained and marked for retry. Every five seconds the server will attempt to resend messages; after ten failed attempts the server will disconnect the client process. The disconnection is an indication to the client, once it recovers from whatever caused its halted state, to reconnect and continue its activities.

Another feature of the client / server package is automatic reconnection of clients to servers. In the event of any disconnection of the logical link between client and server, the client will continually attempt to reestablish the connection every 60 seconds.

The *Alarm Server* task is the central point of the *Significant Event System*. It runs continuously as a detached process on one of the Host system processors. All *significant event* and heartbeat messages are directed to the *Alarm Server*. It distributes all new messages to any clients that have requested such. The *Alarm Server* also monitors all heartbeat messages from critical processes and will internally generate a *significant event* indicating the failure of a process should its heartbeat cease. The *Alarm Server* maintains an internal list of all devices currently in a bad state as indicated by a *significant event* message; hence any newly connecting client process can be informed of the complete state of the experiment. In conjunction with the *significant event* message, the *Alarm Server* also stores a record indicating whether a bad condition has been acknowledged; this record can be generated either manually by an *Alarm Display* task or automatically within the *Alarm Server* by the receipt of *a significant event* which is more fundamental. An example of the latter is the 'off' condition of a device, which is more fundamental than *significant events* associated with individual attributes such as voltages and currents being out of tolerance.

The *Alarm Logger* application is a receiver of *significant events* and thus a client of the *Alarm Server*. It writes each *significant event* message as a single record in a sequential file. In order to avoid filling disk files with oscillating devices, the *Alarm Logger* actually delays writing the record for 60 seconds; if a subsequent message arrives in that time with the opposite state transition for the identical device, then the pattern is altered so as to eventually record only the first and last messages of the sequence along with the appropriate counters. An accompanying user task, the *Alarm Scan*, provides an SQL-like interface to the log files; for example a user may specify a time period and a device name to examine its history.

Another receiver of significant events is the *Alarm Watcher* application. This task explicitly requests that only messages above a certain priority level be transmitted. The priority threshold chosen is that associated with *significant events* that affect the quality of data. The *Alarm Watcher* maintains an internal queue of 'bad' messages; upon the transition from zero to greater than zero this task sends a message to the DAQ *Run Control* task to pause further data acquisition. This action is announced to the operators via a DECtalk speaker. Once corrective action has been taken (which should clear any 'bad' condition) then the DAQ operator manually continues the run, also entering log information which is eventually used to construct a downtime report.

The *Alarm Display* task is the principal user interface to the *Significant Event System*. It is also a receiver of *significant events*. Users may request only specific messages by specifying a set of filter condition groups, or may receive all *significant events*. A graphical display, constructed using the MOTIF windowing system, categorizes *significant events* into 'bad', 'acknowledged', and 'good' messages for each filter group and presents summary counter buttons. The user may select any such counter button to get a list of devices that have generated the messages. From this list window the user may further select a single device for numerical and textual information or launch a parameter page control application for the device. To supply the operator with detailed information on the device, the *Alarm Display* uses the database identifier encoded within the *significant event* message as the key to making a database access. The operator may also choose to acknowledge the significant event by entering an identifying comment. A message indicating the acknowledgment is returned to the *Alarm Server*, which generates a new *significant event* propagated to all potential clients.

The main database for the operation of the DØ control and monitoring software utilizes DEC RDB. It was found that applications directly accessing the database suffered in performance when opening the database for use, and also required significant process resources to work effectively. As a result, a *Database Server* application was created from the same set of client / server tools. Tasks accessing the database are linked with a library of client routines that send messages to a server task, the server accesses the database, and the results are returned in a message. The *Database Server* task is given substantial priority and resources, so as to centralize such needs in a single process.

IV. OPERATION

We present here an example to illustrate the operation of the *Significant Event System*. Consider the case where a critical device goes out of tolerance. The Front End monitoring this device will generate an ACNET protocol, high-priority *significant event* message and multicast it on the Token Ring.

A Gateway task will recognize this message as belonging to the group functional code requested by the *Alarm Server* task and enter the message into the input buffer for that circuit. As the input buffer is processed, a data format conversion occurs. The resulting message is entered into an output buffer for DECNET transmission to the *Alarm Server* on the Host VAX cluster.

The *Alarm Server* receives the incoming DECNET message and places it on an input queue. As the message is processed the internal state record of the experiment within the *Alarm Server* is updated. Client processes are checked to see if *significant event* messages of this type have been requested; if so, the *Alarm Server* transmits the message either by local VMS mailbox or remote DECNET connection.

One receiver of the message is the *Alarm Watcher* client, which is selectively monitoring high priority *significant events*. The message is added to its internal queue; if this is the first such message then the *Alarm Watcher* commands the *Run Control* process to interrupt data acquisition. A DECtalk voice indicates the pause of the DAQ system.

The operator, having been alerted by the DECtalk message and the pausing of the run, interrogates the *Alarm Display* for the cause. A new entry has appeared in the category(s) associated with this particular *significant event*; the operator selects the appropriate buttons and list items to determine the nature of the problem. The original fault

can be corrected, resulting in a 'bad to good' *significant event* being generated, or the operator may choose to acknowledge the fault and continue running. The original 'bad' *significant event* message is superseded by the new 'good' or 'acknowledged' *significant event.* As the operator resumes the DAQ system, a log entry is made of the fault category.

All of this activity has also been transmitted to another client, the *Alarm Logger*, which has written the messages to a disk file. Users may interrogate the log later to determine the circumstances surrounding the fault.

V. SUMMARY

The *Significant Event System* has been used actively at DØ since 1992. It has proved flexible with the ease that member applications can be created or existing applications have functionality added. The system has also proved to be robust, surviving the vagaries of detector hardware failures, network interruptions and application errors. The set of display and logging utilities has provided the experiment's operators with key tools.

In the future DØ will likely base more of its control and monitoring activity on products shared with the HEP community. Many of our current products will be updated to fit within this more general scheme, while retaining those features we have found particularly useful in the experimental environment.

ACKNOWLEDGMENTS

The authors wish to acknowledge the numerous contributions of the Fermilab Accelerator Division Controls Department for their development and support of the Front End and Token Ring systems. We also thank the members of the DØ collaboration for their feedback and contributions to the *Significant Event System.*

REFERENCES

[1] S. Abachi *et al.*, Nucl. Inst. and Meth. **A338** (1994) 185.
[2] A. Ahn *et al.*, Nucl. Inst. and Meth. **A352** (1994) 250.
[3] R. Goodwin *et al.*, Nucl. Inst. and Meth. **A352** (1994) 189.
[4] R. Goodwin *et al.*, Nucl. Inst. and Meth. **A293** (1990) 125.
[5] R. Goodwin *et al.*, Nucl. Inst. and Meth. **A247** (1986) 107.

Alarms and Limits at the Collider Detector at Fermilab

N.S. Lockyer
Department of Physics
University of Pennsylvania
Philadelphia, Pennsylvania 19104-6396

I. What is CDF ALARMS?

The Collider Detector at Fermilab (CDF), is a large high energy physics experiment containing about 20 different detector systems with a total of some 100,000 High Voltage channels. These channels are monitored and can also be set remotely using a Windows interface system called the Alarms, Limits and Remote Monitoring System (ALARMS). The CDF ALARMS is a node on an accelerator-wide network known as the Accelerator Controls Network (ACNET) system.

The Windows interface consists of several loadable menus known as parameter pages that can be accessed via mouse on any of the consoles that are part of ACNET. One index page, the Experiment (E)-page, is devoted to CDF. Once accessed this page has all ALARMS operations on several similar sub-pages. Every operation (Primary Application) has a separate name such as PA1060 for the CDF High-Voltage Control program and PA1076 for the Alarms Monitor, and has an entry on this index page.

CDF ALARMS provides the interface between the on-shift personnel and control and monitoring of the detector. ALARMS is fully integrated into ACNET, and therefore allows the monitoring of the machine status at the beginning and during beam stores as well. ALARMS is not read out in the "fast" data stream.

II. Main Functions of CDF ALARMS

There are three primary functions of the CDF ALARMS system: high voltage control, online monitoring and data logging. A brief comment on each is given below:

- CDF ALARMS is the primary mode of High Voltage Control and online status display for detector systems. The appropriate sub-page is accessible through the E- index. The High Voltage Control function can be used to set individual detector channels, segments of a certain detector system or an entire detector system.
- CDF ALARMS provides online monitoring and display of the HV status (on, off, or standby), the status of low voltage power, temperatures and the detector systems. A summary of set alarms is provided as well.
Low voltage supplies are in general only monitored by ALARMS, although there are a few exceptions. Temperatures, pressures, flows etc. are continuously monitored by ALARMS.
- Data-logging and time plots for luminosity and various detector functions are also plotted on the CDF ALARMS consoles.

III. Hardware Components for the System

The CDF control room contains three ACNET Consoles (Fig 1) and two dedicated monitors for displaying detector status. The Consoles are presently VAX workstations running the VAX/VMS operating system. Any X-station can run the ALARMS program and display the various plots pertaining to detector status.

The heart of the monitoring system is two "i386" Front End processors that continuously monitor the detector from the first floor of the B0 assembly building. The ACNET consoles communicate with the Front End processors via ethernet, and the Front End processors communicate with the Central Database and Operations VAX via ethernet as well.

For the purpose of monitoring low voltages, temperatures, and pressures the Front End communicates with CAMAC-based modules directly using a Kinetic Systems 32- channel Scanning ADC. High Voltage control is either through direct communication to a CAMAC-based HV Controller (CAEN System) or via a series of PCs running a dedicated program monitoring several subsystems. One system (VTX) is controlled using a DAC. The Fastbus system of CDF has dedicated hardware to convert power supply currents, voltages, temperatures etc. to voltages which are connected to scanning ADCs located in CAMAC nearby.

IV. Database and Alarms Setting

Each device or channel that is monitored by ALARMS has a unique ACNET database entry that includes reading, setting and status properties. The database entry also contains a brief description of the device and information necessary for the front end to control it. The program DABBEL is used for database entries and is supported by the Accelerator Division. Only a few designated experts have access to this database.

Alarms are handled by an application known as AEOLUS running on the ALMOND cluster in the Accelerator Division. The Front End reads the hardware value for each channel and compares to its database of acceptable ranges. If the channel is out of tolerance, the Front End sends an alarm report to AEOLUS which forwards it to the Consoles. The appropriate primary application (PA1076) at CDF receives this information, organizes the results and displays them on the monitor.

V. Alarms Display Page

We display three types of Alarms: normal, ignored, and severe. Severe alarms require immediate action and are triggered if crucial channels are out of tolerance. Data taking is stopped and cannot resume until the problem is fixed; severe alarms are announced by DECTALK as they appear.

Normal alarms indicate a channel that may have drifted out of tolerance, but data taking can continue. However if a large number of normal alarms appear, then a severe alarm is set, and data taking is inhibited.

Ignored alarms allow the shift personnel to be reminded that a number of problems do exist but that these can be fixed later. Normal alarms can be ignored by clicking on the alarm. Severe alarms cannot be ignored.

The Alarms display page also indicates the status of various subsystems, the time, and the number of events being processed from AEOLUS. A window at the bottom communicates directly with the user; warnings, errors and informational messages appear here as well.

VI. Summary

The CDF ALARMS system has been running successfully for several years and no major upgrades are planned for it. The collaboration will continue to use the system through Run-II, or roughly the year 2002.

VII. Acknowledgments

The CDF Alarms system is the result of a combined effort by the University of Pennsylvania and the Fermilab Accelerator Division. We would like to thank Kevin Cahill, Mike Glass, Brian Hendricks, Bob Joshel, Boris Lublinsky, Jim Smedinghoff and Junye Wang from the Fermilab Accelerator Division and John Yoh and Jim Patrick from CDF. At the University of Pennsylvania we would like to thank S. Osher, J. Gonzalez, F. Azfar, J. Heinrich and O. Long.

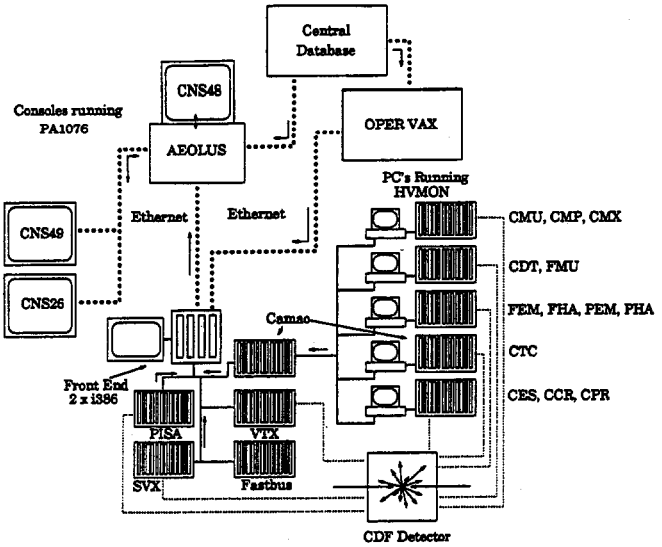


Figure. 1. A schematic of the CDF ALARMS system showing the consoles, Front End, and control of the detector systems.

Beamline Control and Data Acquisition at the Advanced Photon Source

Mark Rivers
Center for Advanced Radiation Sources
The University of Chicago

The Advanced Photon Source (APS) at Argonne National Laboratory is a third generation synchrotron radiation source, optimized to provide high-brilliance x-ray beams from undulators and wigglers. The beamlines at the APS are being constructed by Collaborative Access Teams (CATs), composed of members from universities, industry and national laboratories. The CATs will be conducting experiments in a wide variety of areas, including protein crystallography, microspectroscopy, high-pressure research, small-angle scattering, and microtomography.

Although the CATs are organized and funded independently, and have a diverse set of research interests, they have decided to work together in developing software and hardware for beamline control and data acquisition. These efforts include voluntary hardware and software standards such as:

- Common low level interface for controls
- Agreement on hardware devices to be developed and supported
- Common "look and feel" from beamline to beamline
- Data file formats
- Standard applications for spectroscopy, diffraction, etc.

At the lowest level the EPICS control system will be used to communicate with I/O devices, which will be largely VME-based. The distributed nature of EPICS and the device abstraction it provides permits a variety of higher-level applications to be used. These high-level applications will be specific to various techniques (imaging, diffraction, etc.), but the goal is to provide a common look-and-feel, and a common framework within which applications can cooperate. Many CATs will be using commercial packages, such as IDL and Visual Basic, for the user-interface. As CATs develop software they are sharing it with the EPICS collaboration and with each other through a software exchange. Because EPICS was originally designed for accelerator control, rather than data acquisition, the beamline development groups have added a number of new features to EPICS.

A draft specification for a common data file format, base upon the Hierarchical Data Format (HDF) from NCSA at the University of Illinois has been prepared. Recently the neutron diffraction community has expressed serious interest in using the same standard.

There are a number of important goals of this collaborative arrangement:

- Minimize the duplication of effort in developing control and data acquisition systems.
- Provide a common look-and-feel to users who may use several beamlines for their experiments.
- Ensure that both applications and data are portable between beamlines.

(A related paper on this topic will be published in Reviews of Scientific Instruments, as part of the 1995 Synchrotron Radiation Instrumentation conference proceedings. See "Beamline Control and Data Acquisition Software" by T. M. Mooney, et al.)

FERMILAB-Conf-96/069

Overview of Beamline Control and Data Acquisition at ESRF

F.Epaud on behalf of the ESRF Computing Services .
ESRF
BP 220, F-38043 Grenoble Cedex
FRANCE

Abstract

The ESRF provided photons to beamlines at the end of 1992. Two and a half years later, 14 beamlines (+ 4 CRG beamlines) are open to users and regularly receive visiting scientists. 6 new beamlines are in their final commissioning phase and will be operational early in 1996. This rapid building schedule has been achieved by different small teams using exactly the same technologies for all the beamlines, including the CRG (Collaborating Research Groups). VME front-end hardware and software originally designed for the machine control system, based on the so called 'device server model', have been reused. This model uses client/server communication via RPC and runs on OS-9 (and LynxOS) on the VME front-ends and Unix workstations. In the data acquisition domain, we are currently extending this model to define a 'Modular Data Acquisition Software' to handle the acquisition memories (AM) that are used to collect events or images. The main parts of this system are the 'online display' used to visualise and evaluate rapidly the quality of the raw data collected inside the AM, and the 'fast data transfer' which sends large amounts of data (of the order of several hundreds of mega bytes) at high speed to a central computer facility (NICE) which has a large storage, computing and soon graphics capabilities. The transfer can be effected over Ethernet or ATM. The centralised computing facility (NICE) has been set up to allow short-term data archiving and data treatment by means of dedicated file servers using RAID disk storage arrays, optical and magnetic tape robots for data migration and backup and a loosely-coupled workstation cluster for data analysis and number crunching.

1 Introduction

It is planned to install 30 ESRF beamlines open to the public by the end of 1998 to which 12 Collaborating Research Group (CRG) beamlines will be added [1]. Today, 14 ESRF and 4 CRG beamlines are operational and regularly receive visiting scientists. By the end of this year, 6 new ESRF beamlines will be commissioned, which will increase the total number of open beamlines to 20 and the number of end stations to 24.

The choice made by the council to commit the ESRF to build the beamlines instead of having too many built by the CRG, as it is the case at many other institutes, allowed us to use the same technology on all the beamlines and enables rapid provision of usable instruments to the scientific community. Most of the CRG beamlines also use the ESRF technology.

Both the beamline control and data acquisition use the so called 'Device Server Model' derived from the 'Standard Model 91' defined at ICALEPCS 91, and originally designed for the 'Machine Control System' [2, 3]. This model has been extended for data acquisition needs by the definition of a 'Modular Data Acquisition Software' [4, 5] to handle the acquisition in a standard way for most of the ESRF systems.

On the back-end, the ESRF also provides a centralised computing facility (NICE) for short term data storage and data analysis. Transfer of data is normally made over Ethernet but ATM is also used for beamlines providing large amounts of data.

2 Beamlines architecture

Each ESRF beamline has its own control system and its own network, in order to run autonomously. A router allows communication with the rest of the ESRF and the outside world and also allows access to the 'Machine Control System' through a gateway, to get parameters like machine current intensity, insertion device setup, etc. A dedicated X terminal is connected to the private machine network to control the insertion devices.

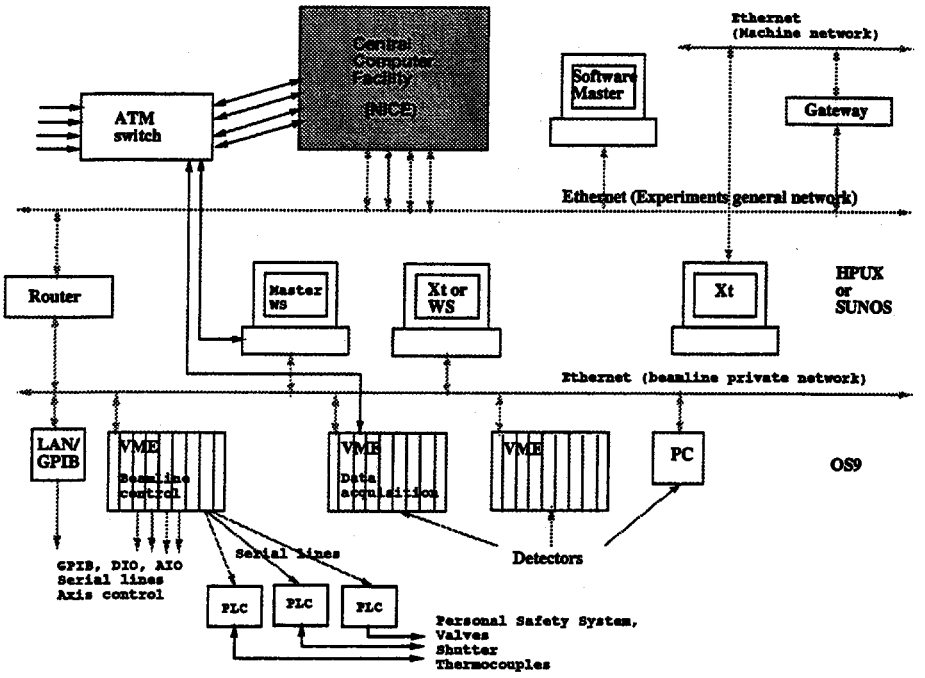


Figure 1: Beamline architecture

A typical beamline is composed of a main UNIX workstation (HP700, Sun) running X11/Motif GUI or a CLUI (Command Line User Interface) client like SPEC [8]. The workstation is used for the beamline control system and often for data acquisition. Other UNIX workstations or X terminals are also added for preliminary data analysis.

On the lower level are the VME systems using the Motorola MVME167 68040 CPU running OS-9, used for instrument control.

Generally at least one VME crate is used for the control of the optics components, vacuum, slits, attenuators, etc. and at least one other crate is used for data acquisition. For beamline control, these crates drive a large number of serial lines, digital or analog inputs/outputs and axis control.

We also use many GPIB devices by means of LAN/GPIB converters directly connected to Ethernet. This choice has proven to be the most reliable one.

The security aspects are left to the PLCs, which are in charge of the vacuum and Personal Safety System interlocks. They are accessed by the VME via serial lines.

PCs are also used as stand-alone systems, mainly to run commercial acquisition systems, such as multichannel analyzers, CCD cameras, image plate scanners, etc.

The philosophy adopted at ESRF is not to keep the collected data on the beamline system but to transfer them to a central computer facility (NICE: Network Interactive Computer Environment). This can be reached by the experiment Ethernet backbone network through the router. However, for beamlines collecting large amounts of data, the transfer rate was too low and overloaded the network. Therefore ATM optical links rated up to 155 Mbits/s have been set up. On some beamlines which use large histograming memories in VME, the crate is directly connected to ATM by means of a VME/ATM adaptor.

Last but not least, a software master to compare beamline software against a reference has been set up. This recently installed mechanism is of great help to keep all beamlines at the same software level and to detect potential problems at an early stage.

3 The Modular Data Acquisition Software

The central part in most of the ESRF acquisition systems is the acquisition memory (AM) into which raw data are acquired. Usually this memory is in the front-end (VME or VXI crate) and has a typical size of the order of several hundreds of megabytes.

Processes dealing with this memory can be divided into two groups. In the first group are so-called producer processes, which define memory organisation and put data into the AM. In the second group are so-called consumer processes, which nondestructively access the AM.

The data structures reflecting the organization of the AM are kept in another memory area usually located on the CPU card, and shared among the processes involved. It is called the AMS (Acquisition Memory Structure).

To minimize the amount of software to be written for different acquisition systems we have defined an API [6] (MEMAPI, MEMory Application Programming Interface) which allows the software to access memory structures (partitions, images, ROIs, pixels) in a uniform way, in total abstraction of the physical arrangement of the data within these memories. Another part of the software dealing with the AM is the data access layer [7] (DATA-ACC) which is the only hardware dependent part needing to be rewritten or ported to every new acquisition system.

Data acquisition control processes that are data producers are processes that are specific for each acquisition system, and in most cases they are written in the form of a device server. Client processes which communicate with the acquisition control process usually run on a UNIX workstation. Their interactive user interface can be either in the form of an alphanumeric menu, X11/Motif graphics users' interface or SPEC application [8]. Additionally we have written two general consumer applications, which are the online display, and the data transfer process.

In order to visualize data while they are being acquired (histogrammed, listed or accumulated), the online display process runs in the front-end CPU. This is an X11/Motif client [9] which, in its current version, allows the display of 2D image data in snapshot or live mode, can show successively a series of images in a cyclic fashion, can pan and zoom, do Z-slicing and X or Y cuts (1 or more pixels wide). Typically, we achieve in live mode a rate of 1 image/second using an MVME167 CPU running OS-9 and an X-server on an HP or SUN workstation. We are also currently evaluating a solution based on an ELTEC-EUROCOM17 bi-processor card (2 x 68040) having embedded graphic capabilities which allow the X11 server to run locally and access the Video RAM via external hardware to refresh the image in real time.

During or after the acquisition run, data are transferred to a central computing facility. The data transfer process, written in the form of a device server, uses the XFRLIB library [10], which contains functions manipulating BSD sockets above TCP/IP. On the physical level, we use Ethernet as well as ATM, either on workstations or on VME crates. For the VME connection to ATM we use a dedicated MVME167 CPU running LynxOS and handling TCP/IP and the ATM driver.

Due to the system modularity and the tightly coupled communication mechanism based on VME shared memory, the processing power can be distributed over several processors. For example, the acquisition system control could be performed by a MVME162 CPU located in slot 1 (which is also in charge of the crate control), a second ELTEC E17 CPU with OS-9 could be used to run

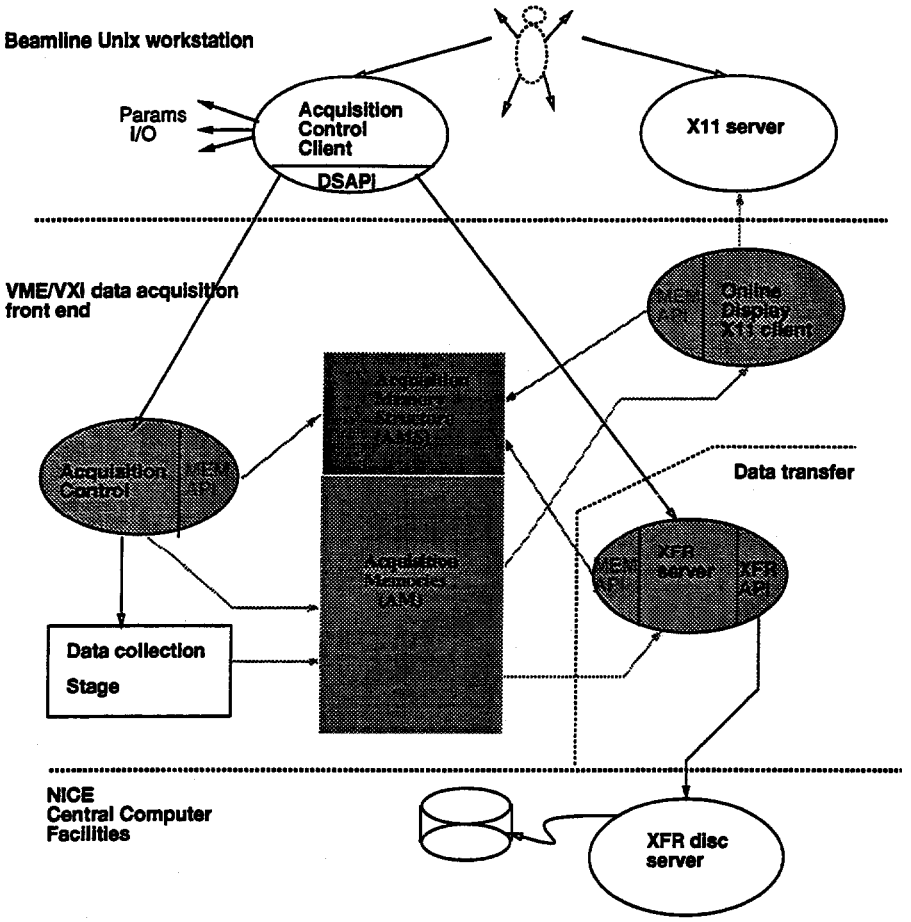


Figure 2: Software architecture of a data acquisition system.

the the X11/Motif online display client and a third MVMEM167 CPU could be used for the data transfer running LynxOS, the TCP/IP network stack and the ATM driver.

This architecture is now implemented or under implementation on all the data acquisition systems designed within the ESRF Computing Services, such as the X-ray image plate scanner, Multi-Wire Chamber Gas-Filled Detector [5], CCD camera, etc. Of course, ESRF beamlines also use commercial systems controlled by PCs or UNIX workstations.

4 NICE: Networked Interactive Computing Environment

The large amount of data collected by the beamline end stations are stored after being transferred, and can be analysed on the central computer facility (NICE) [11].

Due to the volume of data, it is not possible to archive all of them, therefore we have been obliged to fix a limit of 100 days, after which the data are automatically deleted. In addition, visitors normally go back to their home institutes with their data and delete them on NICE before leaving the ESRF.

The 100 days policy is achieved by a three level file migration facility providing a total of 1.5 tera-bytes storage capability consisting of:

- 200 giga-byte RAID7 disk arrays.
- 180 giga-byte of rewritable optical disk.
- 1200 giga-bytes of Exabytes tape robot.

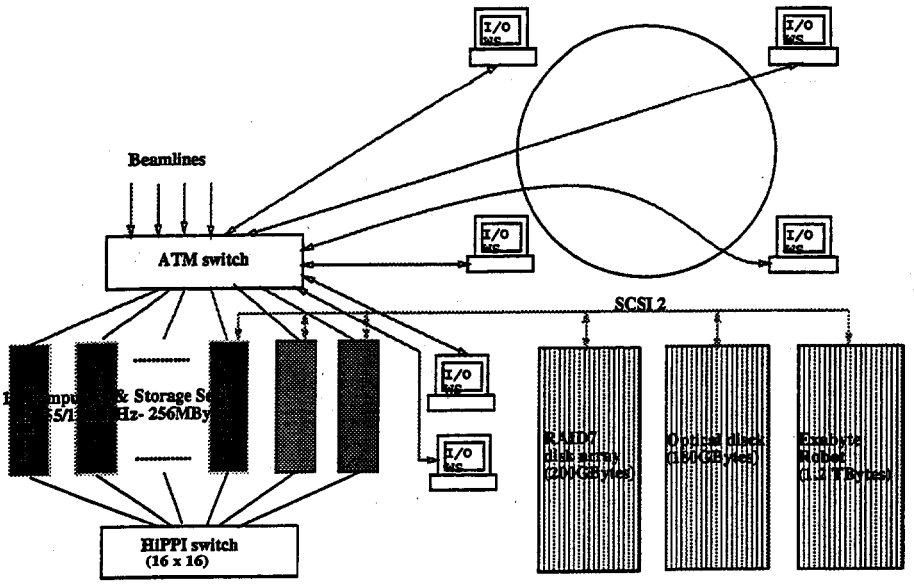


Figure 3: NICE architecture

Data analysis can be performed on a HiPPI cluster of 10 HP755-125, each equipped with 256 mega-bytes of RAM which can be used individually or as parallel computers.

All these computers, including the ones used to control the storage system, are linked with a HiPPI bus (High Performance Parallel Interface) with a bandwidth of up to 800 Mbits/s. Each computer is configured such that a user sees exactly the same environment when he logs on any one of them. Powerful graphics capability and an increase in the number of stations on the cluster are currently under evaluation.

Six I/O workstations, connected with a 155 Mbits/s ATM links to NICE are also available around the ring to allow the users to make their own DAT or Exabyte tapes before leaving the ESRF.

5 Conclusion

The rapid building of the 14 first beamlines has been made possible by reusing the improved hardware and software technology developed for the 'Machine Control System'. We are currently working hard to install the 16 new beamlines scheduled to be operational by the end of 1998, which will all use the current architecture. Even though a great deal of manpower is dedicated to beamline installation, we are constantly improving our 'Object in C' model and are currently introducing C++. Investigations are under way to use ORBIX from Iona (based on CORBA from the OMT) and/or OLE from Microsoft in order to migrate to a system of distributed objects.

6 Acknowledgements

The author wishes to acknowledge the contributions of all the members of the Computing Services.

References

[1] ESRF Highlights 1994/1995. ESRF internal report, September 95.

[2] A.Götz, W.D.Klotz, J.Meyer: Object Oriented Programming Techniques Applied to Device Access and Control in International Conference on Accelerator and large experiment Physics Control System, Tsukuba, November 1991.

[3] A.Götz et. al: Experience with a STANDARD MODEL' 91 based Control System at the ESRF. Proceedings of the International Conference on Accelerator and Large Physics Experiments Control Systems. Berlin, November 93.

[4] F.Epaud, F.Sever, A.Beteva: Modular Data Acquisition Software. ESRF internal report, reference: DASR006, January 1995.

[5] F.Sever, F.Epaud, F.Poncet, M.Grave and V.Rey-Bakaikoa: Modular Data Acquisition System and its Use in Gas-filled Detector Readout at ESRF. Synchrotron Radiation Instrumentation 1995 (SRI95), October 95.

[6] F.Sever, F.Epaud and M.Grave: Memory Library Application Programmers' Interface (MEMAPI). ESRF Internal documentation, reference: DASS003. 1995.

[7] M.Grave: Data Access Programmers Interface (DATA-ACC). ESRF Internal documentation, reference: DASS007, 1995.

[8] Certified Scientific Software: Xray Diffraction Software. Cambridge. MA.

[9] F.Poncet: Online Display Users Guide. ESRF internal report, 1995.

[10] F.Epaud: Sockets Based Data Link Users Guide (XFR). ESRF Internal report DASUG001, 1994.

[11] R.Dimper, B.Lebayle, WD.Klotz: Technical Specification for a Central Computing Facility. ESRF Internal report, October 1993.

A Proposal to Move from the LEP Topological to an LHC Functional Control System Architecture

Michel Rabany
CERN / AT Division, 1211 Geneva 23

Abstract

Our knowledge of the fundamental particles of matter is governed by the available particle energy in accelerators. In order to address this issue, and physicists have been designing ever larger accelerators and colliders for the last forty years engineers. This has led to increasing complexity of both the components and the control signals that have to be relayed to and from an operations centre. Whereas simple cabling was more than sufficient for the early accelerators, distributed control systems are now essential for large accelerators. At CERN, such developments were pioneered by distributing computers around the SPS accelerator, which were then interconnected through proprietary communication links. Networking with an emerging Token Ring was a major breakthrough for the LEP collider. While the evolution of accelerator control systems has matched the progress of technology, industry has, for its part, developed very attractive generic solutions to meet the needs of manufacturers for automation, control and supervision. The clear advantages of industrial control solutions in today's economic climate lead us to consider their integration into modern accelerator systems. Following an analysis of the actual LEP control system, an evolution of its architecture is proposed for the control of the LHC.

INTRODUCTION

During the 1960's, the ever increasing need for data processing was an incentive for the development of computer technology. Experimental physics is certainly one of the most demanding applications in this respect. Concurrently, the ever increasing need for energy in the collision of particles leads to the development of ever larger accelerators. Accelerator control requires communication with the equipment from an operations centre, in the same way that musicians in an orchestra need their conductor. The economic necessity of reducing the numbers and the sizes of the cables to reasonable dimensions led to today's distributed computer control architecture. Our pioneering activity in this domain sheltered us from the outside evolution, steered by industrial needs. The benefit we can expect from using industrial controls certainly requires adaptation of our control architecture for optimal integration. Since 1990, CERN has taken an increasing interest in industrial controls for many applications, which leads us to believe that they will be extensively used in LHC.

THE EVOLUTION OF THE MACHINES

Created in February 1952, CERN started operating its first proton and nuclear accelerator, a Synchro-Cyclotron of 600 MeV, in 1957. The machine, essentially a magnet, was 5 m in diameter. At the end of the 1950's the PS, a circular proton accelerator of 26 GeV for which CERN was built, delivered its first beam. The PS circumference is ~630 m. Then the ISR was constructed in 1971, the world's first proton collider, of 63 GeV and 940 m in circumference. Next came the SPS, a 450 GeV proton synchrotron accelerator, launched in 1976, 7,000 m in circumference and, finally, in 1989 LEP, a 110 GeV electron-positron collider, 27,000 m.

Control of the equipment inevitably relies on cables for the transmission of electrical signals.As the machines were small enough, laying cables from the equipment to a central control room for the SC and the PS was not a problem. The construction of the ISR and the upgrading of the PS at the end of the 1960's provided the opportunity to introduce the emerging computer control technology within a centralized computer architecture. The size of the SPS called for pioneering distributed computer control architecture, in order to keep the computers near the equipment. The result was a system based on Norsk Data ND100 computers, connected in a TITN star configuration [1]. The LEP, the world's largest collider, also depends upon a distributed computer control architecture, but the computers are connected to a Token Ring running around the accelerator [2] (Figure 1).

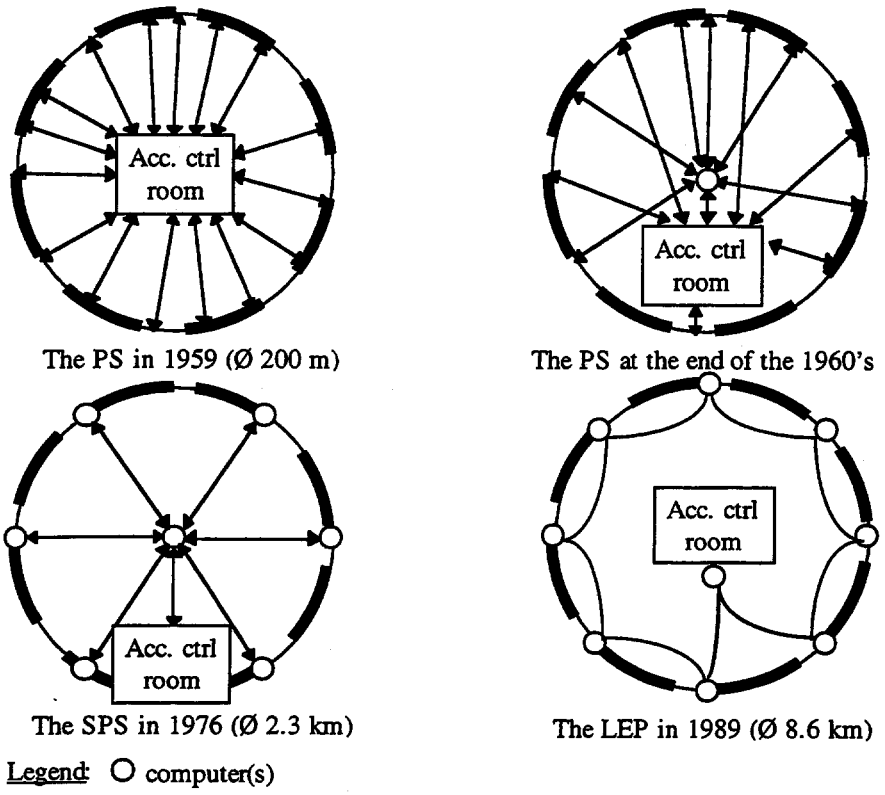


Figure 1. Evolution of accelerator basic control architecture.

THE EVOLUTION OF INDUSTRIAL CONTROLS

The industrial products

Industrial controls really took off with the advent of Programmable Logic Controllers. PLCs appeared in the United States around the year 1969 in response to the automotive industry which wanted to develop automated production lines that were able to follow the evolution of the models [3]. This led to the development of products from two of the largest manufacturers, Modicon and Allen-Bradley. Europe came in two years later with products from Merlin-Gerin and Alspa. The first aim was to replace the cabled solutions which were very rigid and costly. The first specifications for the PLCs included operation in a harsh industrial environment, simple implementation and low cost.

PLCs are designed to work in a harsh environment caused by three main types of aggression:
- physical and mechanical: vibrations, shock, humidity, temperature, etc.
- chemical: corrosive gas (chlorine, hydrogen sulphide, sulphur oxide, etc.), metallic dust, etc.
- electrical: electromagnetic and electrostatic interference, etc.

PLCs are designed to provide a simple tool for the user.
- The design of an application is made easy by the presence of a programming console, which is matched to the mind and the needs of the process technician. Attaching the software to the hardware is a simple task of filling in parameters. The modularity of the hardware and the flexibility of the software allows for easy reconfiguration.
- The operation and supervision are provided through specific consoles. These consoles are easily configurable through application enabling programs, without programming, to perform a wide variety of functions:
- PLCs can provide
    - graphics screen editing
    - alarm monitoring and logging
    - control
    - data acquisition

- report generation
- real-time and historical trending
- data analysis
- help for maintenance
- etc.

- Large installations are controlled by Distributed Control Systems which, in addition, offer:
  - access to multiple clusters of PLCs through LANs or peer to peer communication links
  - distributed applications running on different platforms

The industrial users

It is difficult today to find an industrial sector that does not use automation. The following industries account for two thirds of the market: motor, metallurgy, textile-wood-paper, chemicals-petroleum and energy. The last third is split between mechanics, machine-tools, agro-food and the tertiary sector. [4]

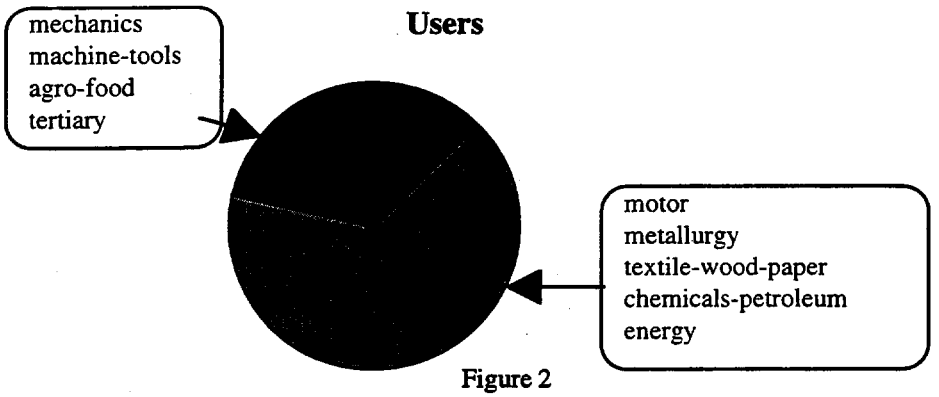


Figure 2

The manufacturers

The world-wide market for PLCs was estimated at $2.5 billion in 1987 and is approximately $5.5 billion today. The growth rate for revenue is around 10% per year, with Europe and the United States being the largest PLC markets. According to Siemens [5] in 1992, as well as stated in Control Engineering [6] in 1995, Siemens and Allen-Bradley were manufacturers number one and two for sales . Mitsubishi ceded its position in third place to Schneider, mainly because of company regrouping. The market for Distributed Control Systems is still led by their inventor, Honeywell (1975), followed by ABB, Yokogawa, Bailey and Siemens.

The world market shares for PLCs and DCSs are illustrated below.

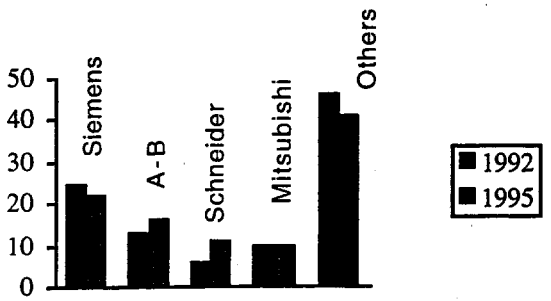


Figure 3. - **World PLC market share**

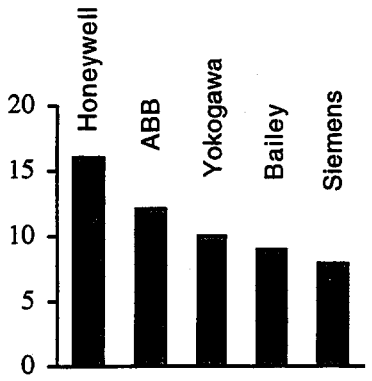


Figure 4. - **World DCS market share**

CONTROL NEEDS OF AN ACCELERATOR

An accelerator is made up of equipment that can be classified into two categories:

- the utilities which are characterized by their capacity to supply a product or provide a service. In this category, we find
  - the production and distribution of:
    - water: raw, drinking, cooled, chilled, heated, demineralized, waste, etc.
    - electricity: high voltage, medium voltage, low voltage, security network, no-break supply, 48V security network, etc.
    - air: ventilated, compressed, conditioned, heated, exhausted, etc.
    - gas: nitrogen, helium and gas mixtures for detectors, etc.
    - cooled helium: liquid, fluid, superfluid, etc.
    - etc.

  The evolution of the physical processes which are involved is slow in comparison to the particle production process of the accelerator. These utilities have a very simple interface with the equipment of the accelerator. Their process is of the continuous type. It consists of maintaining a few specific physical parameters within limits that have been specified with the objective of not affecting the particle production in the accelerator. These parameters are, for example, temperature, pressure, voltage, etc. Industrial actuators and sensors are their control elements.

  - and the services for:
    - access control
    - fire detection
    - radioprotection
    - etc.

  These are made of sequential processes which do not contribute to the particle production in the accelerator. They are slow processes. Industrial actuators and sensors are their control elements.

- the production equipment which is characterized by direct involvement in the particle production. In this category, we find
  - the electromagnetic equipment and their power supply: dipoles, quadrupoles, sextupoles, kickers, etc.
  - the electric equipment: radiofrequency cavities
  - the electrostatic equipment: deflectors, separators, etc.
  - the beam instrumentation: electrostatic pick-up's, beam scanners, intensity monitors, etc.
  - etc.

  The particle production process involves a perfect synchronization between the thousands of pieces of equipment spread along the accelerator. The current of the power supplies and the

electric field in the radiofrequency cavities are tightly coupled. The physical phenomena which govern the production of particles are fast. Electromagnetic, electric and electrostatic equipment are the actuators and beam monitors are the sensors. They are not industrial. Industrial actuators and sensors are not their control elements.

THE LEP CONTROL SYSTEM

As illustrated in figure 1, the LEP control system is a distributed computer control system. The circumference of LEP has been split into eight equal sectors, in order, for one thing, to have reasonable distances to supply the equipment with what it needs in order to perform its functions. The control system fits closely with this structure and acts like a funnel for data coming from, and commands going to, the equipment.

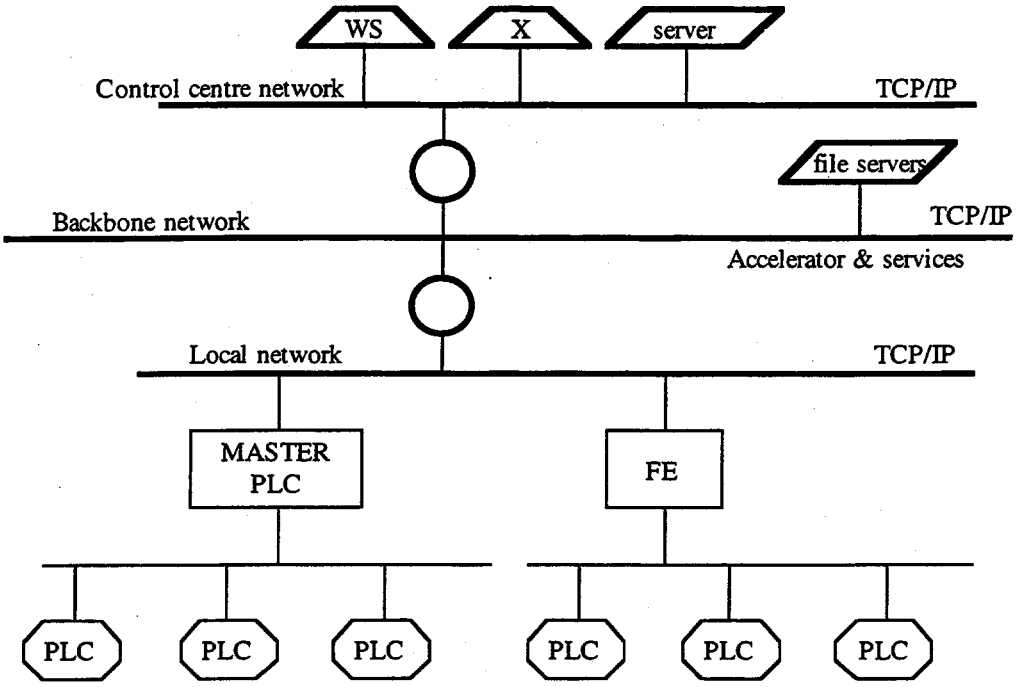


Figure 5. Integrating PLCs today in the LEP Control system

Figure 5 shows the possibilities today of integrating PLCs in the LEP control system. It must be recalled that the particle production equipment described above is not considered here.

The Front End computer [7] [8] or controller imposes a very rigid frame structure which makes the original structure of the equipment control layer completely disappear. A direct consequence is the difficulty of integrating an industrial Distributed Control System which the manufacturer has designed to give a central global view of the equipment. It is much easier to use PLCs at a lower level, that is at network level. However, integrating Master PLCs into the network and reconstructing the supervisory functions at the level of the accelerator control room can only be done at the expense of considerable effort. In addition, this structure has the serious drawback of dividing into two parts the responsibility of the people in charge of the equipment. This is due to the topological structure.

The pressure to use industrial controls could be satisfied by the possibility of a more global integration. Splitting the equipment into functional domains (centres) would immediately remove the inconvenience of breaking the split of responsibility and allow for much better integration of industrially available control solutions. Moreover, this structure fits much better with the requirements concerning the flux of exchanged data or commands for the types of equipment concerned. The specific Centre may apply a constructive filtering to the whole flux of data coming from the equipment to the benefit of the accelerator operation. Figure 6 shows the resulting functional arrangement.

The organization to functional elements is something which, interestingly, can be adapted to the human resources structure. The Water Centre, in order to fit with distributed responsibilities, may itself subdivide in more specialized Centres like Waste Water Centre, Chilled Water Centre, Raw Water Centre, etc., as there is no intrisic reason to restrict the water processes to a single Centre.

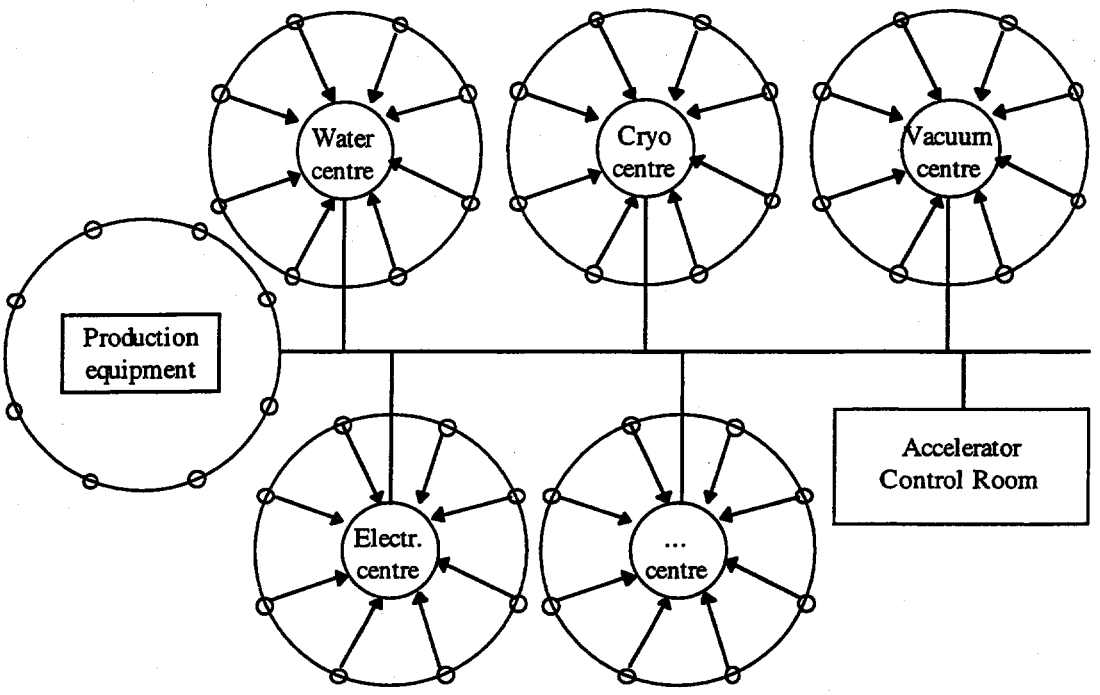


Figure 6. The functional approach

CONCLUSION

The vital necessity for our research laboratories[9] to make as much use as possible of industry leads us to adapt our concepts to the industrial world. The industrial world is naturally organized into specific domains which match the specialization of the individuals from whom expertise is expected. Tendering for equipment on the basis of functional specifications will result in industrial offers from which we can only expect the maximum benefit if we do not impose restrictions on the overall system. This naturally leads to a functional architecture.

REFERENCES

[1] The design of the control system for the SPS, M. Crowley-Milling, CERN 75-20.
[2] The LEP Control System, P.G. Innocenti, Accelerator and Large Experimental Physics Control Systems, Vancouver, Canada, 1989, pp 1-5.
[3] Les A.P.I., Architecture et applications des automates programmables industriels, G. Michel, Dunod edition, Paris, 1988
[4] Quelques chiffres sur les API, Robotique Informatique Industrielle, G. Rouchouse, Janvier 1992.
[5] Le marché des API en 1992, Robotique Informatique Industrielle, Mars 1993.
[6] Market update, G. J. Blickley, Control Engineering, January 1995.
[7] Interfacing industrial equipment to CERN's accelerators and services control system, P. Anderssen, P. Charrue, R. Lauckner, P. Lienard, R. Rausch, M. Tyrell, M. Vanden Eyden, CERN-SL-95-42 CO.
[8] SPS and LEP Controls: Status and Evolution Towards the LHC Era, R. Rausch, This conference.
[9] Industrial Control Solutions in Research Laboratories, M. Rabany, This conference.

20 MeV Microtron Control System

J.S. Adhikari, Y. Seth and B.J. Vaidya
Center for Advanced Technology, Indore, INDIA

INTRODUCTION

The Centre for Advanced Technology (CAT) was set up in 1984 at Indore by the Department of Atomic Energy. The major thrusts at the CAT are in the areas of accelerators, lasers and related research and technology. The main aim of the accelerator program is to develop particle accelerators for research and development and for medical and industrial applications. As part of this program the 450 MeV Indus-1 and the 2 GeV Indus-2 electron storage rings were completed as Synchrotron Radiation Sources (SRS). The corresponding critical wavelengths of the emitted radiation spectrum are 61 A and 1-4 A respectively. Both the machines are supplied with electrons by a common Booster synchrotron (20 - 700 MeV). A 20 MeV Microtron was chosen due to its simplicity as the injector for the booster. The main design parameters of this microtron are a pulse current of 30 mA and a pulse duration of 1-2 μs with repetition rate of 1-2 Hz. This injector microtron was commissioned in 1993. This paper describes the control system for it.

REQUIREMENT

The microtron being used at CAT is a dual-purpose machine. It is used as an injector to the booster and also it is a part of a free electron laser system. Its control system is basically governed by the following requirements:

-The operation of the microtron should be controlled both centrally and from a local control room.
-The subsystems of the microtron which are to be monitored and controlled are the magnet power supply, the RF modulator and driver, the cathode power supply, vacuum, beam diagnostics, cooling system, vertical probe movement for scanning of different orbits, beam extraction, field measurement, quadrupole focusing and defocusing power supplies, analyzing magnet power supply, timing and trigger system, alarm, safety and interlocks.
-The system must work in noisy environment, noise generated by the RF modulator and other sources.
-The operation should be fail safe. It should monitor all safety aspects of machine, operator and working personnel, as the activity level is high.
-It should be reliable.
-It should be user friendly.

ARCHITECTURE

The entire control system of Indus-1 is designed around the VME-based Supervisory Control And Data Acquisition System (SCADAS) and PC/AT computers functioning as operator consoles. It is designed as two layer system. The upper layer consists of a number of PC/ATs on an ethernet. The control system architecture is modular and distributed because it is divided into a number of subsystems, each of which is partly autonomous. The microtron control system is a part of this with the ability to be controlled from two locations.. The overall system is shown in figure 1. Each console is backed by an individual SCADAS based on the VME architecture. All the interfaces have been developed in-house to avoid any future maintenance problems. The industry standard VME architecture was selected as its specification satisfies the requirement of the accelerator control environment. It is a 32-bit bus with the capability of handling a multi-CPU environment, bus arbitration, interrupts and many others useful features. The SCADAS consists of following modules:

-CPU module: This has a 16 bit 68000 processor, 32 kB RAM, 32 kB EPROM, serial communication port and parallel port.
- 32 bit optically-isolated input module: This is used for scanning the status of different units.
- 32 bit relay output module, used for controlling various units.
- 32 channel, 12 bit ADC card with memory.
- 8 channel 16 bit DAC with hardware trimming facility and voltage to current converter.
- 8 channel signal conditioning card (current to voltage converter).
- 8 channel stepper-motor controller card.
- serial interface card to connect the field measuring unit with any other system.

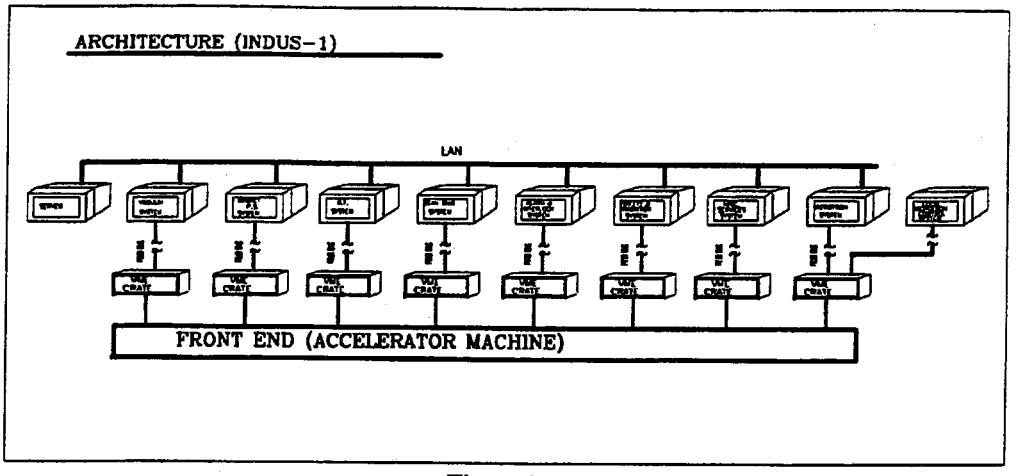


Figure 1

SCADAS is housed in a 19 inch rack located at the local control room together with an RF synthesizer, a digital CRO, a video monitor, a status monitoring and display system, the timing and trigger system, a PC/AT as a console and radiation monitor, safety and interlock systems. SCADAS is connected serially to the console through one of the communication ports. The microtron control system is shown in figure 2.

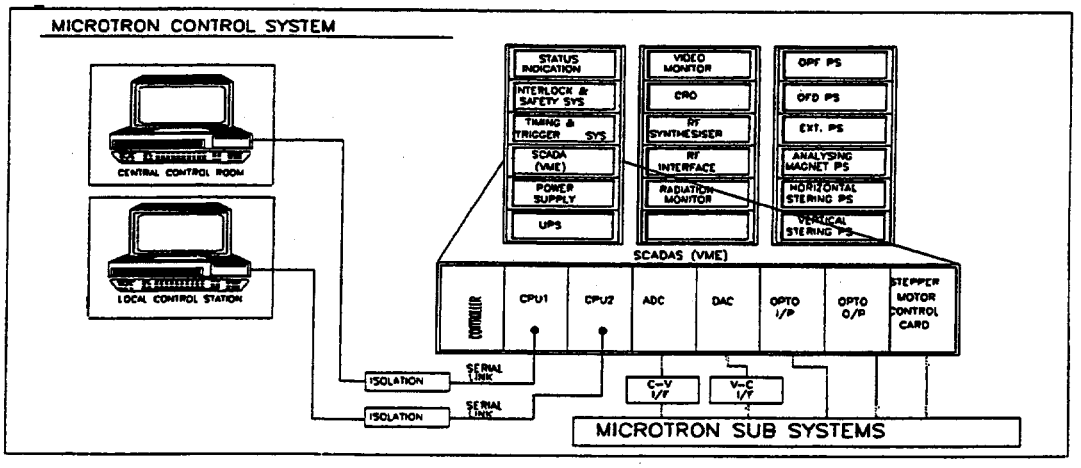


Figure 2

OPERATION

There are two modes of controlling the microtron, for commissioning and for normal operation. Some interlocks are bypassed during commissioning to obtain the results in the shortest possible time. This is allowable, as experts are handling the machine during this period, but interlocks must be in use for normal operation. In the initial phase of commissioning, we started with a probe, driven by a stepper motor, raised in the vacuum chamber to check the beam orbit by orbit, from the first to the 22nd. In a later phase, the probe was removed and the extraction channel was mounted, with a fluorescent screen at the end, viewed by a video camera. The current pulse was viewed on a digital oscilloscope. The beam was extracted after aligning the channel parallel to the beam. There are quadrupole lenses in the straight beamline to adjust the beam size according to requirements.

The vacuum required is 10^{-6} torr or better. The power supply for the heater of the lanthanum hexaboride cathode is interlocked with the vacuum, to prolong its life. RF power is provided by a frequency synthesizer, amplifier and modulator. The cavity is tuned by varying the synthesizer, manually when operating from the local control room and through GPIB from the central control room. The cavity temperature is stabilized to within 5^{o} C to avoid excessive frequency shift.

There are three important parameters to control in a microtron; RF power, magnetic field and emission current. The settings of such parameters are entered digitally by the operator at the console. These values are converted by 16-bit DACs into analogue currents in the range 4-20 mA. These are connected with shielded cables to the units to be controlled, where they are converted to voltages in interface boxes. Isolation amplifiers have been used for the important parameters to avoid corruption of the signals by noise, mostly coming from the nearly 130 kV RF modulator pulse. Parameters are monitored by converting the signals to currents in the 4-20 mA range in interface boxes. After transmission to the central control room, the currents are converted into voltages by a signal conditioning cards at the SCADAS and then converted by a 12-bit ADC. On/off commands are carried out by relays and the system and interlock status is sensed via opto-isolators.

SAFETY AND INTERLOCK

For normal routine operation a sequential procedure has been implemented to achieve safe and accident-free operation. There are two main aspects of safety; machine safety and personnel safety. All individual units are interlocked according to requirements such as water flow, temperature, overvoltage, overcurrent, door open etc. The system interlocks such as vacuum, radiation level, air ventilation etc. affect the machine operation. All the system interlocks are monitored continuously and some of the critical ones are executed in hardware. For human safety we have installed radiation dosimeters, a search-and-scram system inside the microtron and booster hall, an alarm system, a public announcement system, a siren and door interlocks.

TIMING AND TRIGGER SYSTEM

The complete system of Indus-1 is synchronized with the 50 Hz AC line. The timing scheme is shown in figure 3. The zero crossingr pulses are derived from the cathode power supply and divided to obtain 1 or 2 Hz, which are the microtron operating frequencies. Firstly ramping of the booster dipoles is started then the septum magnet power supply is switched on. The time to switch on the modulator in the microtron is chosen such that the magnetic field at the septum has leveled out and the dipole current is at the injection level. The modulator pulse is energized during the zero crossover slot. The current through the cathode in this period is minimum, to avoid modulation of magnetic field in the vicinity of cavity when rf field is applied. The timing and trigger system is interlocked with the safety system for proper operation.

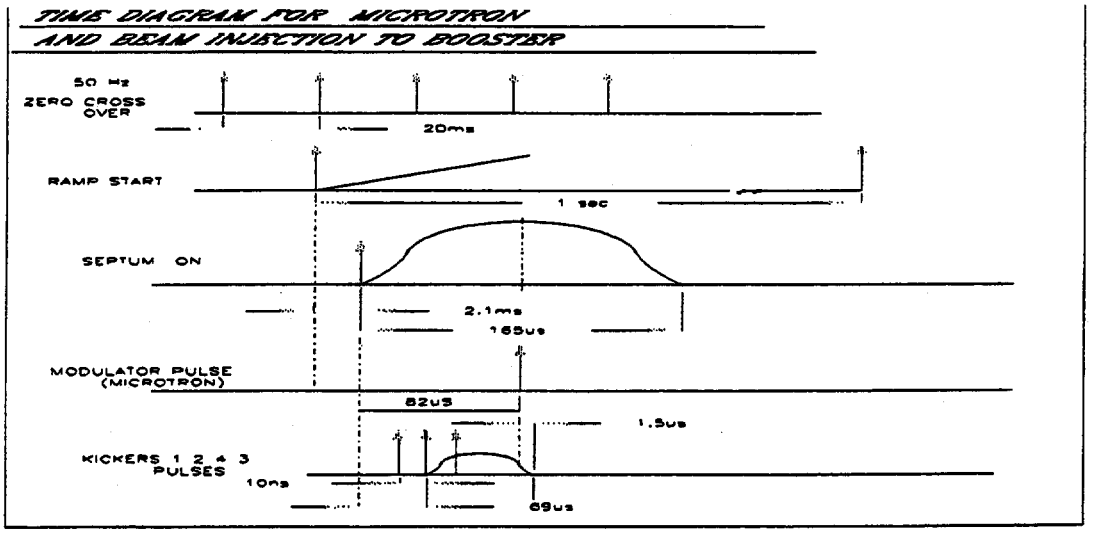


Figure 3

SOFTWARE

The main task of software is to supervise the overall system activity. The software for the operator interface has been developed using the Turbo C package under DOS. Software for SCADAS is written in assembly language using a cross assembler and then downloading. The interface was made user friendly for ease of use by the operator.

CONCLUSION

The microtron control system has been working reliably at the Centre for Advanced Technology for the last three years. There are occasional problems due to modulator noise due to the deterioration of the RF grounding. Recently we have commissioned a 12 MeV microtron system at Mangalore University for research and development. Work is going on to build a microtron for medical and industrial use. A similar control system will be used for with them.

REFERENCES

1. Kapitza S.P. and V.N.Melekhin, The Microtron, Physics Laboratory of the Institute for Physical problems, Academy of Sciences of the USSR, Harwood Academic Publishers, London, Chur, 1978.
2. Waldmar Scharf, Particle Accelerators and their Uses, part-II, vol. 4, Harwood Academic Publishers, New York, 1986.
3. Safety Report, Microtron, Linac and DC accelerator Testing facility, Centre for Advanced Technology, Indore, India, August, 1989.
4. 20 MeV Microtron- Project Report, H.C.Soni, M.M.Bemalkhedkar and S.S.Ramamurthi, Centre for Advanced Technology, Indore, India, 1989.
5. Control System for Synchrotron Radiation Source (INDUS-1), B.J.Vaidya, International Conference on Current Trends in Data Acquisition & Control of Accelerators, VECC, Calcutta, INDIA, Nov. 19-21, 1991.